This is to certify that the

thesis entitled

PERFORMANCE DATA MODELING, TRANSFORMATIONS, AND MULTIPLE-DOMAIN ANALYSIS METHODS

presented by

Abdul Waheed

has been accepted towards fulfillment
of the requirements for

M.S. degree in Elect. Engr.

Diane Hwede Rover

Major professor

Date 11/11/93

PERFORMANCE DATA MODELING, TRANSFORMATIONS, AND MULTIPLE-DOMAIN ANALYSIS METHODS

BY

Abdul Waheed

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Electrical Engineering

1993

ABSTRACT

PERFORMANCE DATA MODELING, TRANSFORMATIONS, AND MULTIPLE-DOMAIN ANALYSIS METHODS

BY

Abdul Waheed

Observed data is regarded as a useful source of analysis for systems involving a great deal of complexity in terms of their physical structure and functioning. A parallel computer system is also a system of this type with a parallel program having the processors work concurrently and interact with one another to solve a problem. Therefore, parallel program monitoring and visualization based on the data observed and collected during the execution of a program are considered usable and effective methods to analyze the performance of the system and program. Conventional performance visualization and analysis methods and tools can provide information about the higher-level program abstractions to the programmer, but are often less effective when the performance data volume and complexity increase due to an increase of problem and/or system size. We have developed a matrix based model to represent, process and analyze the performance data to more rigorously navigate through this complexity. Matrix transformations are applied to a performance data matrix to transform the data to spatial, temporal, event, and frequency domains, for the purpose of better visualization and analysis focused on specific aspects of the program. Specific aspects of program behavior are analyzed using appropriate analysis techniques that are prevalent in statistical signal processing and digital image processing, and this in turn, contributes to the understanding of the macroscopic behavior of the whole program. This approach has been prototyped by using commercially available data analysis tools and conventional performance visualization tools, and it meets the objectives of scalability, extensibility and efficacy for a variety of program performance analysis applications.

To the Creativity

ACKNOWLEDGEMENTS

The author acknowledges the invaluable assistance and guidance of everyone who has been involved with this work. The accomplishment of this work would not have been possible without Dr. Diane Rover's guidance of the research work, patience to listen to the author's ideas and encouragement for applying them to an entirely new area. Moreover, this thesis could not have such an excellent flow without her meticulous editing. The author is grateful to Dr. Hassan Khalil and Dr. Majid Nayeri for their contributions to author's vision and understanding of the related areas of this multi-disciplinary research work. Dr. Richard Enbody of the Department of Computer Science deserves special thanks for providing general evaluation of this work, despite his very busy schedule.

Some of the nCUBE-2 programs used for the case studies presented in this thesis were originally written by Edgar Kalns of the Department of Computer Science, who deserves our appreciation for allowing us to work with them.

An internet survey was conducted to assess the use of matrices in related areas of computer science. The author acknowledges the contributions from all the respondents. Relevant publications of the respondents have been appropriately referenced.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The analysis of program performance on parallel computer systems is a non-trivial task due to the complexity of their architectures and the programming paradigms implemented on them. Conventional methods that have been used for the performance analysis of sequential computer systems and programs often do not contribute towards enhancing our understanding of the behavior of parallel systems and programs. Several approaches have been proposed to tackle the performance evaluation problem of a specific parallel system using a specific performance analysis method or tool. However, we are still far from a comprehensive methodology to take up the task of performance evaluation of any program executed on any parallel system. This thesis is an effort to define certain aspects of this problem by appropriately characterizing the performance of a parallel computer system, and then to use this characterization to develop an analysis approach that could help analyze certain aspects of this task in a generalized manner.

Program monitoring and Performance visualization are considered useful techniques for the analysis of parallel program performance. These techniques depend on the performance data obtained by executing the instrumented parallel programs on parallel systems. However, these methods are limited by several factors including their scalability, extensibility and lack of rigor in their analysis methodologies. The objective of our approach is to develop more rigorous methods for analyzing performance data visually as well as statistically, which are scalable and extensible and which represent some physical and/or logical structure of the parallel system and program. This approach supplements and complements program performance analysis on parallel systems using visualization.

In the rest of this chapter, we will discuss the motivation for this work and the analysis approach used for this purpose. Then a brief introduction of the performance analysis

domains, transformations, and analysis methods will be provided before considering them in detail in subsequent chapters.

1.1 Motivation

We, as engineers and scientists, have learned to adopt certain approaches to analyze various systems or phenomena of varying degrees of complexity. Characterization of complex systems and behaviors by appropriate parameters (metrics) and modeling and analyzing them by available mathematical tools is a commonly followed practice. However, characterizing the system behavior by certain metrics and building up an accurate mathematical model of the system does not guarantee a realistic as well as practicable analysis. If mathematical model is built on simplifying assumptions, it makes the solution of that model simple but we have to compromise its accuracy in practical situations. On the other hand, if we do not make simplifying assumptions and build up an accurate model, its solution for simplest of problems will be non-trivial, and therefore, the model will be impractical for more complex problems. Such models that represent the physical structure of the system components (usually in the form of differential/difference equations) have been referred to as *structural models* [24]. These models use a collection of state variables in vector form, $x(t)$, to represent the state of various elements of the system. The state variables are related with the dynamics of the system model using vector differential equations of the form:

$$x'(t) = f(x(t), U(t), t) \tag{1.1}$$

where $U(t)$ is an input vector, $x(0)$ are initial conditions, and t is the independent variable. Such models have been solved using analytical methods such as transform techniques, analog computers for more complex systems, and digital computers to numerically solve models of large systems. Digital filters were developed to numerically implement the transfer functions that result from transform solutions to study the structural view of the system.

With the development of digital computers, another view of systems emerged. The models based on this view are concerned with the operations that it performs rather than the structure of system components used to perform an operation. Such system models have been termed as *operational models* [24]. The system is observed only at certain discrete times (called *events*) and is assumed to be composed of a set of processes operating concurrently. Process outcome is considered as a sequence of system states (discrete and continuous) and events (system state transitions). Such models have been used for *discrete-event simulations* of the systems.

We consider a parallel program executed on a parallel computer as two entities of an intricate system whose behavior and performance are to be analyzed. Such systems have been modeled by various mathematical models such as queuing models, petri nets, Markov models, etc. These models tend to represent a combination of what has been termed above as structural and operational views. However, these models are often so intricate due to the complexity of the parallel computer system dynamics, that an average user of the system can not use them for routine performance and program debugging purposes. During the last few years, the practitioners of parallel systems have been able to develop another paradigm for the analysis of the behavior of the systems (parallel computers and programs). This paradigm has been termed as *system monitoring* [27, 28, 50, 86, 89] which views the system in a way that is closer to the operational view. Parallel systems are *instrumented* using specialized hardware and/or software libraries to observe the states of the system when certain events occur due to the interactions of a program with the computer system. This observed execution trace data is *visualized* using state of the art computer graphics facilities that are now available with almost every desktop workstation. Various performance instrumentation and visualization tools such as ParaGraph, Pablo, Seeplex, etc. [28, 50, 86] have been developed that are considered an integral part of parallel program development environments. Although the system monitoring technique has contributed greatly to simplify the process of analyzing the behavior of parallel systems, it has a few constraints. These constraints are listed in the following:

1. The monitoring environments are usually not based on any well-known physical and/or programming paradigms that are often used for parallel systems. The analysis performed by visualizing the observed (instrumented) data is entirely dependent on the type of events generating this data. Although this strategy works in many simple cases, the capabilities of such analysis are limited when the computational (logical) structure of the program is complex.
2. Instrumenting a parallel system generates a very large amount of data for an average sized program which is needed due to the generic nature of displays used for the purpose of program performance visualization. Instrumentation as well as visualization do not consider any specific system and programming paradigm, therefore, it is usually not possible to decide what not to observe. Consequently, long-running real application programs are very hard to instrument by using this technique.
3. When observed data are graphically presented to the user as a time-series plot or in the form of other displays specifically used for performance visualization, the lack of scalability is perhaps the first problem that a user encounters with such tools. The presentation techniques often do not adequately scale with an increasing number of processors in the system or with increasing problem size. This problem is critical for various scientific and engineering computations that are typically solved on large-scale parallel processors and often involve large amounts of program as well as performance data.
4. Most visualization tools are restricted to presenting the *visual* information to the user. This is a much more effective way to assimilate the information by a human user, compared to the presentation of the same information textually. However, the presentation methods used by these tools are not *extensible*, i.e., they do not exploit the analysis potential of any particular representation of the observed trace data.
5. The lack of extensibility often makes the efficacy of the monitoring tool questionable. Program instrumentation is a costly procedure in terms of the time it takes to monitor the execution of various instructions and to log the event-specific data, and in terms of the perturbation of system behavior as a consequence of this intrusive procedure. A user expects effective analysis by the visualization tool after going through the expensive procedure of executing an instrumented program.

One motivation of the work presented in this thesis was to try to overcome the above limitations of the system monitoring paradigm for analyzing the behavior of parallel systems. Although the system monitoring paradigm enjoys the acceptance by a wider cross-section of the user community, this does not mean that we can ignore other system analysis approaches. Most of the limitations of existing methods and tools for parallel system monitoring are due to the lack of any rigorous mathematical model that could be used to make this paradigm work as a generic analysis tool. Therefore, we have accepted the system monitoring paradigm due to its simplicity but are not indifferent to other system analysis approaches that are used for the analysis of parallel computer systems, in

particular and other systems in general. One might conclude from the above discussion that the performance analysts and practitioners have entirely different points-of-views regarding the performance analysis. While one group considers the field a science, the other considers it an art. Consider the following excerpts which all relate to the "art" or "science" of computer system performance evaluation:

> *Formation of models of physical behavior underlies science. Yet our ability to construct behavioral models of parallel and distributed programs is very limited. [1]*
>
> *... the emphasis on the "art" of computer design, of system management, of programming, and so on, is antithetic to the quantitative philosophy of performance evaluation. The performance evaluation viewpoint emphasizes the scientific method, well-planned and controlled experimentation, and careful use of mathematical techniques for prediction. [34]*
>
> *Designing an enlightening animation is truly a psychological and perceptual challenge.... At present, creating effective dynamic visualizations of computer programs is an art, not a science. [13]*
>
> *Contrary to popular belief, performance evaluation is an art. Like a work of art, successful evaluation cannot be produced mechanically. Every evaluation requires an intimate knowledge of the system being modeled and a careful selection of the methodology, workload, and tools. When first presented to an analyst, most performance problems are expressed as an abstract feeling, like a rough sketch, by the end user. Defining the real problem and converting it to a form in which established tools and techniques can be used and where time and other constraints can be met is a major part of the analyst's "art." [109]*

The field, in general, has historically asked, "Is it art?," both for computer architecture [111] and for computer programming [60]. Clearly, performance evaluation based on observing a parallel system's behavior by monitoring program execution on an architecture also faces this question. This is especially true for the presentation and analysis of performance data using visual means.

Parallel system monitoring can present fundamental information about program execution. While it is increasingly being deemed a powerful tool for parallel program performance analysis and debugging, it is typically considered to be more of an art than a science. What a user sees and hears can depend on his or her perception of the graphics and sound employed in the presentation of information. Moreover, the user often is largely unassisted in the analysis of this information. The relationship between what is seen and/

or heard and program behavior is based on sensory perception and knowledge of the program. While this brings invaluable insights into the exploration process, there is a need to assist the user with analysis methods that support a more rigorous approach. With this, the user gains not only more insights, but also the capability to substantiate sensory perception with analytical results. The question is, what are appropriate analysis methods that bring visualization and auralization closer to being a science? We address this question in the context of program monitoring of medium- and large-scale parallel systems. The overall objective of our approach is to develop more rigorous methods for analyzing performance data visually as well as statistically. The methods should be scalable with system and problem sizes, and extensible (so as to facilitate further analysis). Moreover, the results should represent some physical and/or logical structure of the parallel system state.

The parallel system performance analysis approach that is presented in this thesis is founded on the idea of considering all relevant system analysis approaches and available mathematical tools as our candidate resources that should be applied appropriately to solve the problem at hand. As engineers, this is the approach that we have been using to analyze various systems and are aware of its capabilities. We use the instrumentation techniques that are available with most of the monitoring tools, but for the purpose of analysis, we have tried to extend the existing methods using the above approach.

Specifically, this approach to performance analysis has three aspects: (1) modeling of trace data as a matrix for the purpose of analysis, (2) transformations applied to this matrix, and (3) performance analysis using multiple-domains. Subsequent sections introduce the motivation of these guiding principles for the definitions and analysis presented in subsequent chapters.

1.2 A Model for Performance Trace Data

Most of the limitations of existing performance monitoring methods that were mentioned in section 1 are a direct consequence of the lack of any rigorous mathematical structure

associated with the execution trace data. Most of the instrumentation systems log the trace data observed during the execution of a program in the form of a trace file (binary or ASCII text) whose fields are arranged according to their own semantics. This file often consists of *trace records* consisting of the *time-stamp* of an event (where the occurrence of the event results in logging a particular event as a record in the trace file) with the event-specific information. These trace records are used to update the visualization displays to "replay" the execution of a program. A trace file does not have any well-defined mathematical structure which could be used for rigorous development of analysis methods for the data. Some would probably argue that records of a trace file use the structure of a relational database and thus relational calculus (by considering a record as a tuple). This supports the implementation of various information processing tasks and relational data analysis, but not rigorous mathematical analysis explicitly.

When the system size or the problem size increases, the displays of these tools can not adequately scale to accommodate large amounts of information over longer periods of execution time. Some of the tools do use various graphics functions such as scrolling a window with a scroll-bar or decreasing the resolution of the information but such techniques are questionable in two respects:

- The macroscopic context of the information is lost if we scroll a display to accommodate a larger (or longer over time) display, as such viewing of information is typically not coupled with any type of analysis that is performed on the data.
- The resolution of the time-scale of various time-series plots of performance data is reduced in order to accommodate more data in a given display. Some tools (such as ParaGraph) usually discard the events occurring during the minimum step size to accommodate more temporal information. This is an effort to preserve macroscopic perspective but this type of approximation is made by sacrificing the precision and often leads to wrong results/assumptions about the program behavior.

Neither of the above techniques genuinely improves the scalability of the tool. Our experiences with the existing visualization tools and the methods that are presented in this thesis indicate that scalability can not be achieved unless the data structure used for representing performance data is mathematically rigorous. If performance data are modeled by such a mathematical structure then the definition of the representation will not

have to be changed with the size of system and/or problem. As it will be discussed in subsequent chapters, various generic analysis methods can be applied successfully once the underlying data structure conforms to certain rigorous definitions. We have selected matrices to model and represent the performance data.

1.2.1 Matrix Representation

By definition, a matrix is an $r \times c$ rectangular array of numbers that are subject to certain rules; where r represents the number of rows and c the number of columns in the matrix. Matrices are considered a very useful notation to represent numbers and several techniques of matrix theory are used to analyze them in various contexts and disciplines. Particularly, from the systems analysis perspective, matrices and techniques of linear algebra are used to get a *"global"* view of the system and to understand its behavior [99]. The use of matrices and linear algebra is not restricted to the systems sciences only. These techniques are used in almost all those disciplines that deal with the analysis of large-scale systems characterized by a number of different parameters and variables. Matrices serve a dual purpose of presenting an overall *"feel"* for the state of the system as well as allowing powerful algebraic and numerical methods to analyze the system according to specific objectives. A large-scale parallel computer system is not an exception. It has a lot of concurrent activity that affects its overall state which is not easy to understand without powerful analysis techniques. Based on the state information (in consequence of specific events) obtained from such a system, we will introduce a technique (in chapter 3) to represent the system states (defined from event traces) in the form of a matrix and then subject them to the applicable and relevant analysis methods to understand the behavior of the system in response to a parallel program (a sequence of inputs to the system).

1.2.2 Examples from Other Areas

First let us briefly look at the application of matrix theory to some other systems and the analysis objectives in those disciplines. A few examples are presented here to appreciate

the motivation for such analysis techniques that will be applied for the performance analysis of large-scale parallel computer systems.

1.2.2.1 State-Space Analysis of Linear Systems

A discrete-time, linear, time-invariant system can be represented by a set of difference equations consisting of (state) variables that represent the internal state of the system at different locations within the system. This representation became generally known as *state-space* representation of the system [96, 113]. This differs from the conventional representation of the system in terms of the relationship between input and output signals (known as a transfer function). The state-space representation has two main features:

- It represents the relationship between the input and the output of the system, as in the case of the transfer function approach.
- The state-space description is complete in the sense that it gives information about the internal states of the system as well. The state variable representation characterizes the relationship between the internal signals (states) as well as the external signals.

Due to these two features of this representation, a much more complete picture of the dynamic behavior of the system can be visualized. For this type of representation, the whole system can be considered as a "box" with internal states represented by n state variables $v_i(k)$ at discrete time k as shown in Figure 1-1. There are m inputs to the system,

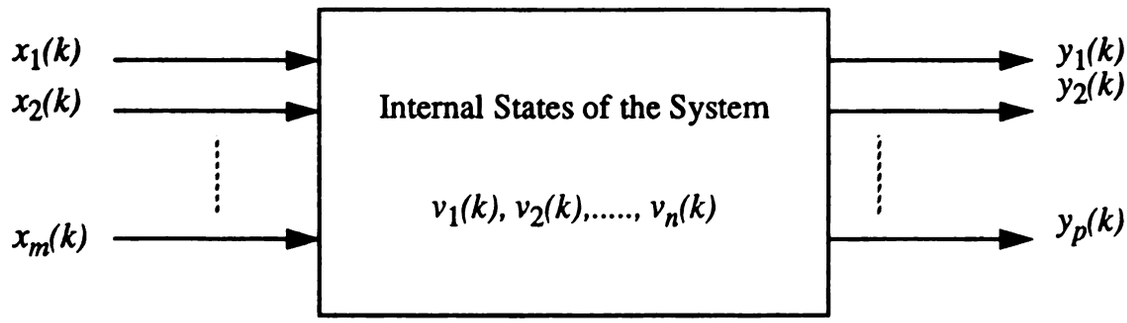


Figure 1-1. State-space representation of a system.

represented by $x_i(k)$ and there are p outputs represented by $y_i(k)$. Such a system is called an n-dimensional linear system (as there are n state variables) and can be represented by two sets of linear equations [130]. The first one represents the states of the system, and the

second represents the outputs of the system as a function of state variables. These are written in the matrix form due to the linear relationship between the system parameters and the state variables. For the system shown in Figure 1-1, the state-space representation is:

$$\begin{bmatrix} v_1(k+1) \\ v_2(k+1) \\ \cdots \\ v_n(k+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_n & a_{n-1} & a_{n-2} & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} v_1(k) \\ v_2(k) \\ \cdots \\ v_n(k) \end{bmatrix} + Bx(k) \tag{1.2}$$

The B matrix is $n \times m$, and x is $m \times 1$ vector representing inputs to the system. The output equitation is given as:

$$\begin{bmatrix} y_1(k) \\ y_2(k) \\ \cdots \\ y_p(k) \end{bmatrix} = Cv(k) + Dx(k) \tag{1.3}$$

where matrix C is $p \times n$, and D is a $p \times m$ matrix. The matrix on the right-hand side in the equation (1.2) is of dimensions $n \times n$ (call it A) and it describes the dynamics of the system under consideration. This completes the generalized state-space description of the system. Once the system is described by this type of a state model, then all the subsequent analysis is based on the four matrices A, B, C and D. As a matter of fact, these matrices provide a lot of information about the behavior of the system. This is accomplished by subjecting these four matrices to various matrix operations (such as determination of eigenvalues and eigenvectors, matrix multiplication, inversion, diagonalization, rank-determination, etc.) to get information about stability, controllability, observability and input-output relationships of the system. The physical realization can be changed by applying various transformations [96] on the given realization {A,B,C,D} of the system. All these analysis and design steps would have been very difficult to perform without using this state-space representation of the system, and still they could not have provided so much insight about the whole system. So, in the case of a generalized system, matrices

helped us to practically "*dump*" a lot of information about the state of the system which facilitates the analysis even when the system size becomes large.

1.2.2.2 Application to Digital Filter Design

In general, a digital filter can be viewed as a system with inputs, outputs and internal states [84]. An N-point digital filter (let's consider a FIR filter here) can be represented by the difference equation showing the input-output relationship as:

$$y(n) = \sum_{k=0}^{N-1} b_k \cdot x(n-k) \tag{1.4}$$

The digital filters are modeled by using adders, multipliers and delay elements. The realization of the digital filter specified by the above difference equation is given as in Figure 1-2 (for N=3). There are two delay elements in this filter (denoted by Z^{-1}),

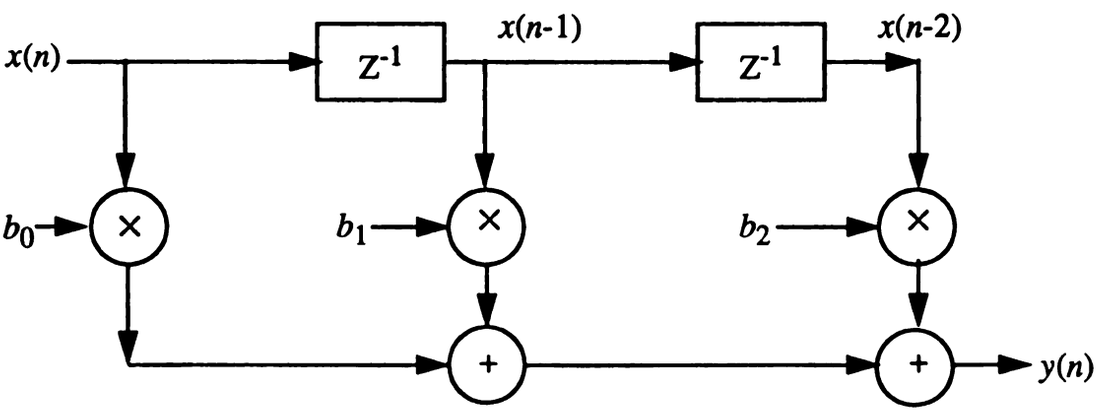


Figure 1-2. Digital filter with N = 3.

therefore, according to the conventions of modeling such systems, the number of state variables that we choose to represent this system is two. An obvious choice of such state variables is to have state variables $v_1(n)$ and $v_2(n)$ such that the overall system can be written in the form of matrices as $v(n+1) = Av(n) + Bx(n)$ and the output can be represented by the equation $y(n) = Cv(n) + Dx(n)$. $v(n+1)$ is a 2×1 vector. A and B are 2×2 and 2×1 matrices respectively, and C and D are 1×2 and 1×1 matrices, respectively. After representing this filter by the state-space model, there are two important design and analysis objectives that can be achieved immediately. These are listed as:

- Useful information such as the stability of the system (filter) can be obtained directly from the eigenvalues of matrix A [84, 96]. Similarly the input/output relationship can be determined from matrices A, B, C and D.
- More important is the capability to transform this filter realization to some other realization by applying some transformation techniques on the matrices A, B, C and D. This is very important in some practical cases. For instance, a filter can be converted to a parallel form (from a tapped-delay form) by diagonalizing the matrix A (and changing B, C and D as well, correspondingly). This is very useful to take advantage of parallel processing. Also the effect of round-off errors is less severe for certain types of realizations, therefore, this type of transformation helps [84].

This was another example of using matrix theory and state-space analysis to get a global view of the system and to analyze it. When the systems become large-scale, it becomes increasingly difficult to achieve the above objectives without using state-space analysis and linear algebra techniques.

1.2.2.3 Simulation

Simulation is an alternative way to analyze the behavior of a system by developing a model for it and then testing this model [64]. The system can be characterized by discrete-time states and events (i.e., the changes in the states). This type of model is used to gain insight into the behavior of the system by observing the real system and then comparing the results from the model under the same types of conditions. This process not only validates the model but also provides useful information about the internal states of the real system which can not be obtained without severely disrupting the operation of the actual system. As an example of the use of matrices as a data representation structure, consider various simulations based on queuing models that often represent the number of customers waiting in a queue at given time and service time for a customer (and other stochastic variables of interest) as vectors. Simulation may be the only realization of a system if the real system is not yet constructed.

1.2.2.4 Representation of Visualization Data

Examples of the use of matrices and linear algebra that we have considered so far have come from systems analysis. In this section, we consider an example that comes from

graphics and visualization. Visualization data in general comprises a multidimensional data set. There is no universally accepted way to characterize the format of this data [15, 112]. Almost every tool has its own data representation and storage format. There are some implicit rules that everyone wants to follow, such as representation of data in ASCII instead of binary format. This is mainly due to the self- documentation advantage of the ASCII format despite the trade-off with performance [86]. The important aspect, however, is resolving the data semantics associated with the visualization data, so that it could be used by various tools. Format conversion of visualization data, whether in a raw form or rendered in two or three-dimensional images, is a practical problem when this data needs to be displayed, analyzed or printed. We only consider the aspect of data representation for analysis purposes.

Visualization data, in general, require various types of analysis functions to obtain insight about the systems where it comes from. The choice of analysis techniques is often dependent on the objectives of analyzing data. These objectives can be so varied, even conflicting in nature, that no single visualization tool can support all types of analyses in an optimal manner. The user may have to import the data to other tools to deal with such situations. This often means an intermediate step of converting the data exported by one

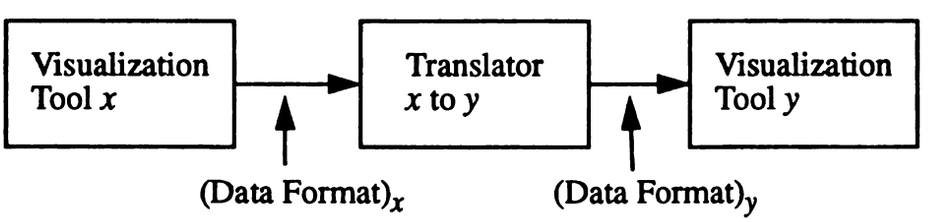


Figure 1-3. Conversion of data for different tools.

tool to another format that the second tool can import and process as shown in Figure 1-3.

One aspect of the matrix approach for performance data modeling needs to be mentioned specifically: despite the powerful mathematical structure that a matrix is for the purpose of analysis, we do not advocate their use for actual processing of data. Researchers [75] have

already tried to optimize computer architecture as well as software development to utilize the matrix as only data structure. But it turns out that this might not be an optimal solution. Other information processing structures such as spreadsheets and database systems might out perform the processing of matrices. Therefore, we model the performance data as matrices only at a conceptual and functional level, such that the approach is beneficial for defining, developing and managing the analysis methods and algorithms. The actual coding of these methodologies or algorithms in specific programming languages and on particular computer architectures need not be restricted to using the matrix as the only data structure, and other more efficient implementations should not be overlooked. The actual implementation of the analysis methods and/or algorithms will thus not only benefit from the rigorous mathematical structure of matrices to develop analysis techniques that could overcome the limitations of existing tools, but also will be able to benefit from efficient implementation strategies.

1.3 Matrix Transformations

Matrix transform techniques are often used when systems are modeled using matrices. In general, transformation methods have always been a key to solving many engineering problems that are hard to analyze in the form (or domain) that they physically evolve. These transformations often result in the availability of mathematical tools and representations of the problem that greatly enhances its understandability, compared to what we have in the original domain where it actually evolved. Linear transformations have traditionally been done with matrices in the context of vector spaces [4, 101], and often result in the transformation of the shape of the problem to another shape (or format) which might have some advantages over the first. The commonly used (linear) transformations include: Fourier transformation to analyze the *spectrum* of a signal by converting it from time domain to frequency domain; *Laplace* and *z-transforms* to design continuous-time and discrete-time systems, respectively; and *similarity* transformation also to design and analyze both continuous- and discrete-time systems by converting the state-space realization of the system from its original form to another form.

Matrices have been used to implement various types of compiler optimizations, particularly to perform loop transformations. [66] describes a loop transformation toolkit that uses non-singular matrices for loop restructuring. The purpose is to restructure the parallel nested loops in such a way that the data dependencies are eliminated to have course-grain parallelism at outer loops. This method is based on transformation from one iteration space (with data dependence at outer loop) to another iteration space that does not carry data dependence. Without such transformations, it is not trivial to determine this type of loop re-structuring.

We have used matrix transformations in the context of modeling the performance data as a matrix. The purpose is the same as mentioned above, i.e., conversion of the performance data to forms that are easier to analyze. Performance data in its original form, even when it is modeled by a matrix, is not very useful for analysis due to a large number of dimensions. Usually, a number of parameters are recorded as fields in a record whenever some event occurs. Some times, more than one record must be logged in response to the occurrence of a specific event. All the fields and/or records related to an event contribute to the dimensionality of the performance data. The key to multiple-domain analysis (as will be explained in detail in next section and chapter 5) is to selectively hide some dimensions of this data, using appropriate transformation techniques. This process transforms the data from "performance" sub-space to another lower-dimensional subspace (typically one- or two-dimensional) which can be analyzed by several available mathematical tools. It will be shown in later chapters that the characterization of the performance data in transformed sub-spaces will further enhance the efficacy of the analysis.

1.4 Multiple-Domain Analysis Approach

Researchers are experimenting with advanced quantitative and qualitative techniques for handling the complexity of performance data. These include the (automatic) identification of phases or patterns in program execution [17, 52, 73], where [17, 3] apply statistical methods often used in signal processing; the use of multivariate data plots [89] and the

identification, organization, and perusal of related data by multivariate analysis [29] and cluster analysis [87]; the use of three-dimensional visualizations [35, 95]; the use of multimedia [13, 39] and virtual reality [86]; characterization in the event and frequency domains [1]; and development of event-based models of parallel system behavior [1, 68]. [65] advocates the use of *multiple-views* i.e., presentation of performance data using several types of displays to understand various aspects of the problem. The consensus is that we need to call upon candidate resources to attack the problem of analyzing and presenting performance data. To this end, we have developed the *multiple-domain analysis* approach [122]. Consider this excerpt:

> *The history of science shows us that a given problem can fruitfully be examined by different disciplines, partly because when it is viewed from new perspectives, implications of alternative assumptions are explored by researchers with different backgrounds or interests, and partly because new techniques developed elsewhere are imported to explore areas left untouched by the discipline in which the problem originated. [5]*

The objective of multiple-domain analysis is to define performance in terms of *analysis domains*, where certain domains have their origin in disciplines other than parallel computer system performance evaluation. Multiple-domain analysis has been developed by:

1. defining a set of domains,
2. characterizing domains via *performance images* and *signals* (to be defined in chapter 4),
3. transforming performance data into usable forms in each domain, and
4. applying visualization and analysis techniques in each domain (using domain-specific tools).

Depending on the types of analyses to be done on the performance data, we have identified four related analysis domains, which can be characterized either by *performance images* [93] or *performance signals* [94]: spatial, temporal, event, and frequency. [109] has emphasized the use of *application domain* to understand algorithm-specific behavior of programs by using application-specific displays and animations. The application domain is of interest as well, but is distinctly different than these general domains. These analysis

domains derive from the multiple dimensions associated with performance data. Each analysis domain focuses on a particular dimension of the data, hiding the details contributed by the other dimensions.

Analysis domains are rigorously defined in chapter 5, but they are introduced here in an intuitive manner. The *spatial domain* emphasizes the range of values of a performance metric over the "space" of the system, where "space" represents some regularly-structured ensemble of processors (or processes). This domain can be characterized by performance images. The *temporal domain* represents the distribution of values of a performance metric during the execution time of a program. This domain can be characterized by performance signals. The *event domain* is similar to the temporal domain, except that a performance metric is expressed in terms of events and not time [1]. That is, the occurrence of an event (a state transition) is significant, not the time between events (the duration of a state). The *frequency domain* also is related to the temporal domain. It shows the spectrum of a performance signal and concentrations of energy at particular frequencies (rates of change of a performance metric). It emphasizes variations of a performance metric and is useful for the types of analyses that will be performed on performance signals. Considering the frequency domain is important, because fast changes often make themselves apparent to us as a rate of change. The event and frequency domains also can be characterized by performance signals. Performance analysis via these domains takes advantage of analytical tools specific to these domains.

The multiple-domain analysis methods are based on strategies typical of scientific visualization, as both scientific visualization and performance visualization share the objective of rendering and projecting large volumes of multidimensional data. These strategies include focusing and linking [14] and the use of image and signal processing techniques to enhance the visual analysis [125]. The main contrast between scientific visualization and performance visualization is that the former renders the natural, physical structure of the system under study, and the latter contends with a hierarchy of structures, including the higher-dimensional logical structure of the system under study. Thus there

are challenges. Effectively applying image processing techniques to performance data relies on representing the performance in at most three dimensions, either rendering the physical structure of the machine or selected aspects of the logical structure of the system (program and machine) (i.e., exploiting logically related system data). For signal processing, accurate results depend on appropriately characterizing the nature of the signal projected from the performance data. Hence, the key to using image or signal processing methods for performance analysis is the meaningful transformation of the performance data into an image or signal. Performance images and signals enhance visual analysis. Images or signals become the basis of analysis and can be compared across architectures, algorithms, and problem sizes. Performance signals also enhance aural analysis, as auralization of program execution is getting more attention due to the multi-media concepts becoming popular in visualization [13, 38]. For example, signal processing techniques—filtering, correlating, etc.—can be applied to aural data to assist the user in understanding what is being heard. Auditory signatures can be reinforced visually with graphs of signals, and the signals can be parameterized and classified. In short, a performance image or signal is to be used hierarchically with other, more conventional views, contributing to a macroscopic display of system state in a particular domain.

The question of interest is: are advanced methods radical departures from what's found in "normal" performance analysis tools? The answer is both "no" and "yes". "No," because tools often already support the presentation of data (e.g., multivariate or time-series plots); but "Yes," because tools typically do not support the analysis of data. So the similarities provide a common denominator between conventional and advanced methods, and the differences help the user answer the questions about performance. Either way, there is utility to the user. For example, views of performance images and signals generated with the standard tools AVS (Application Visualization System) [8, 9] and Matlab (from The Mathworks, Inc.) [71], respectively, have analogs in ParaGraph [50] (such as the Processor Status and Aggregate Processor Utilization displays, respectively). Unlike ParaGraph, the standard tools provide many well-known and tested functions for analyzing the data in the views. Thus, the approach is based on integrating performance

tools and standard data visualization/analysis tools using a matrix representation of performance data in a performance analysis environment as a "toolkit".

1.5 Scope of this Work

In this chapter, we have introduced the motivation of this work in general terms. This has been an inter-disciplinary effort involving diverse areas such as parallel and distributed computing, program performance monitoring and modeling, matrices and linear algebra, statistical analysis (in general) and analysis of stochastic processes (in particular), systems theory, digital signal processing, and digital image processing. Figure 1-4 shows this approach from the perspective of a user of a parallel system. Various disciplines are the

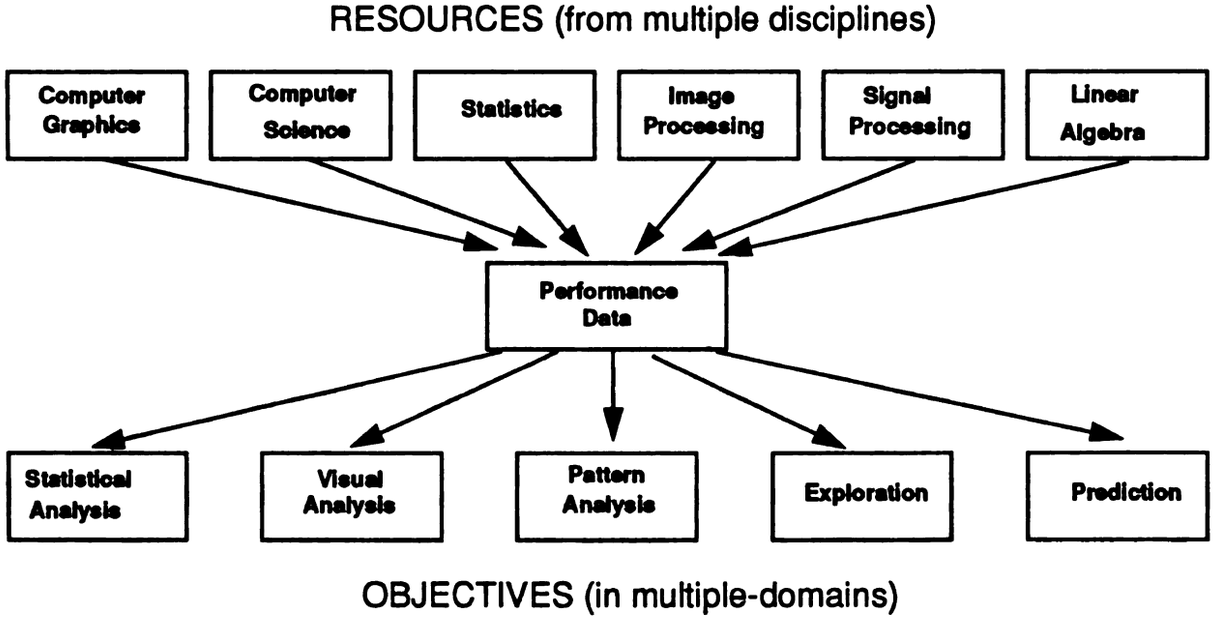


Figure 1-4. Multiple-domain analysis approach from a user's perspective.

source of candidate resources that are called upon to achieve the objectives of visual, statistical, and various other types of analyses that will be presented in chapter 5. An analysis objective, such as performance prediction is a part of the future directions of this research work.

In more specific terms, the primary objectives of this work can be divided into two categories: theoretical and practical. The theoretical objectives will be dealt with

implicitly throughout this thesis, and we will explicitly point them out as they apply to various practical implementations of the analysis methods. The theoretical objectives of this work can are:

1. To overcome the limitations (mentioned in section 1.1) of existing performance analysis techniques and tools.
2. To develop performance data presentation strategies that can enhance the visual and statistical analysis.
3. To glean information about high-level programming abstractions from performance data that typically consists of low-level details of system behavior.
4. To recognize the observed performance data obtained through instrumenting the parallel programs as a realization of a stochastic process (characterization of whose nature remains a future direction of this work). This will lead to the determination of various statistical characteristics of this data, in order to consolidate the theoretical foundation that is the basis of many analysis techniques presented in later chapters.
5. Try to narrow the gap between purely analytical techniques of performance evaluation and purely application-specific techniques.

The following objectives related to the practical implementation of these theoretical objectives have been achieved, so far, by this research:

1. Modeling of performance data by using a matrix approach.
2. Transforming the performance data matrix representation to other lower-dimensional sub-spaces (domains).
3. Applying multiple-domain analysis methods after transforming the performance data to these domains, using statistical signal processing and digital image processing techniques on one- or two-dimensional transformed domains. These analysis methods depend heavily on the matrix representation of the performance data and its transformations to other domains. Therefore, all the analysis methods presented in this thesis are rigorous and hence scalable and extensible.
4. Using generic, commercially-available data analysis and visualization tools to implement the above analysis approach. For the case studies appearing in this thesis, AVS and Matlab have been used for this purpose.

This chapter has presented the context and motivation of the research work presented in this thesis. A survey of the related work will be presented in chapter 2. Chapters 3 to 5 have been used to establish the theoretical framework for the multiple-domain analysis methods. Readers that do not intend to go through the mathematical framework presented in chapters 3 and 4 can go directly to chapter 5 that discusses the multiple-domain analysis

methods, without loss of continuity. Chapter 6 is devoted to case studies to establish the validity of the multiple-domain analysis approach formulated in earlier chapters.

Chapter 2

Related Work

Performance data representation has not been a thoroughly researched area, but a number of performance tools have dealt with this problem, either explicitly or implicitly. Performance data is captured from the execution of a parallel program using available instrumentation options, i.e., hardware or software instrumentation [104]. This is often referred to as "raw" trace data because its format is governed by the semantics defined by the instrumentation system and is not very useful for analysis by the user (other than brute-force textual analysis). Performance analysis tools take this raw trace data and use it to drive their displays which are designed exclusively for the purpose of program performance visualization. Most of the visualization tools (with very few exceptions) use a specific instrumentation system to obtain the trace data that they can process. Therefore, the data semantics are handled within the tool.

We have classified the related work on performance analysis methods, approaches and tools in three categories. These categories are:

1. analytical methods and tools;
2. performance monitoring and tools; and
3. application specific methods and tools.

This classification is based on the analysis methods (or approach) that the tools use, i.e., either generic (formal and rigorous) analysis methods or application-specific methods. Figure 2-1 shows this classification in graphical form. As we move towards the left side of this figure the analysis methods become more generic and mathematically powerful. On the other hand, if we move to the right, the analysis methods become less rigorous and more program specific. We present the related work starting from generic methods and ending with application-specific methods. In a sense, generic and application-specific methods have entirely different approaches to the problem but share the common

objective, i.e., performance analysis of parallel computer systems. Each approach has certain advantages over the other. Parallel system monitoring based methods represent a compromise between the two. Monitoring-based methods depend on measurements of the system parameters when the system already exists, while analytical methods are important to answer "what if" type of questions at the design stage of a system. Application-specific methods are most useful when a programmer could portray an informative picture of an algorithm. There are other classifications of performance analysis methods, based on criteria different from that considered here. For instance, [69] classifies performance analysis methods into three areas: monitoring, benchmarking, and prototyping. Here, the objective is the design of a parallel system and not the performance evaluation of a program on an existing system. Similarly, several other classifications are possible.

We discuss the methods and an overview of the work being done in each of these categories. For each, the first part discusses the approach, and the second part discusses the tools that are being used to implement the approach.

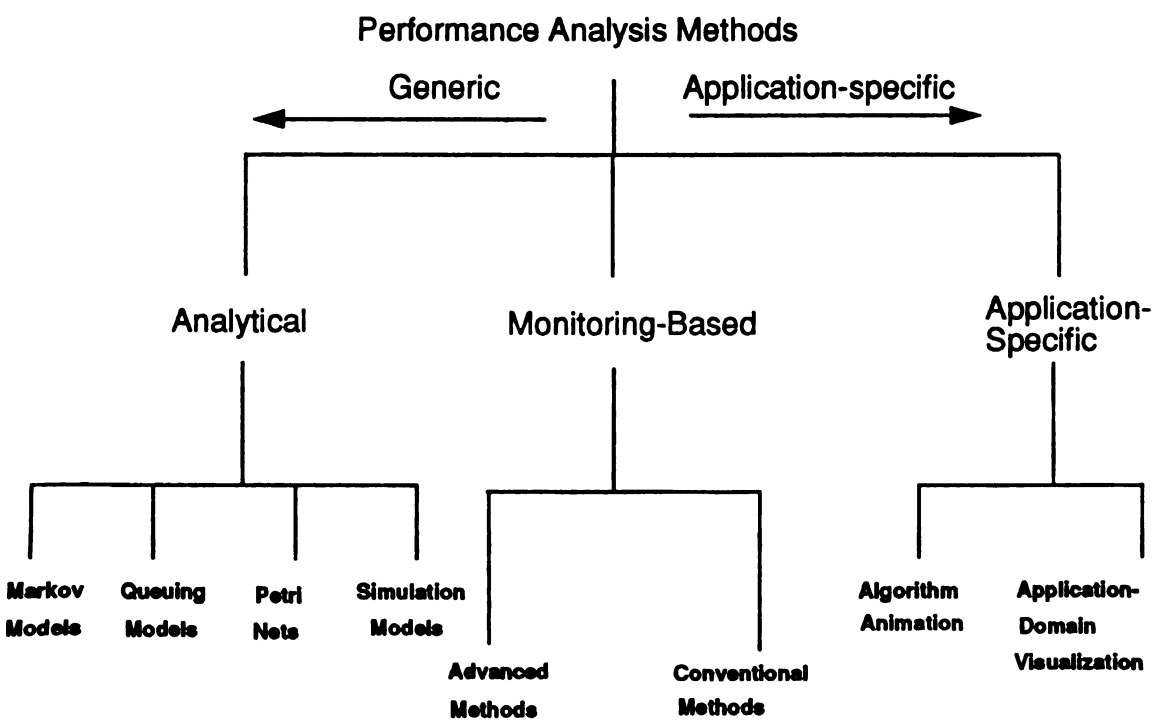


Figure 2-1. Classification of performance analysis methods and approaches.

2.1 Analytical Methods

Performance analysts believe that the pace of developments in computer system design has always overwhelmed the development of adequate and unified theoretical characterization efforts for these systems [33, 56, 69]. The importance of appropriate analysis methods becomes obvious when the system complexity increases. Analytical methods are based on building appropriate mathematical models for the computer system (and computation) to better understand the system and provide insight to the designer. Such models are most appropriate for making various decisions at the design stage of such a system, by analyzing the behavior of the system using alternative architectural choices. The complexity of analytical techniques can result from the abstract nature of the performance evaluation objective, such as, optimizations applied to hardware, operating systems, compilers, networks, and so on. A number of commercially available parallel and distributed systems have been developed under different design and performance goals. Their relative merits are not yet fully understood to enable the system designer to use appropriate models for the system to analyze and predict the performance to compare various design alternatives. Therefore, the evaluation of better performance is still an actively researched problem.

We will survey the discipline of analytical modeling based on the models/methods that are often applied to parallel systems. [33] divides an analytic study of a computer system into three parts: (1) model formulation; (2) model solution (or simulation); and (3) model validation. Analytical models are solved either symbolically or numerically, in order to obtain desired results. A model of a parallel system consists of two parts [69]:

1. description of the architecture, and
2. definition of the workload under which performance predictions are to be obtained.

Figure 2-2 represents a general framework for evaluating analytical models according to the steps described above. It should be noticed that various architectural parameters are expected to behave in a non-deterministic (or random) manner under a given workload. Therefore, a realistic analytical model has to be stochastic, in general. In the following

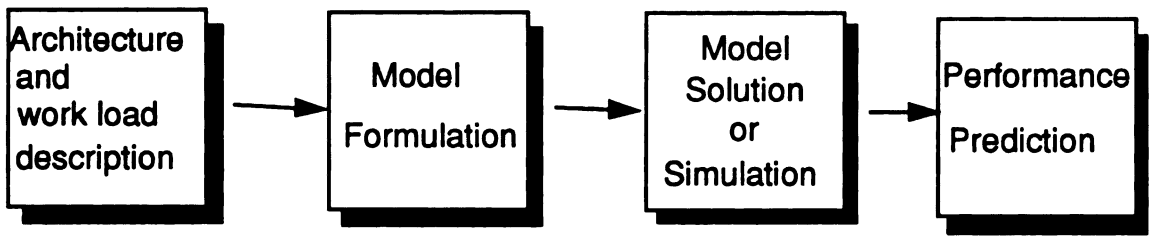


Figure 2-2. Phases of an analytical study of a parallel system.

sections, we will discuss four generic parallel computer system modeling techniques: Markov models, queuing models, petri nets, and simulation models; and survey a few tools based on these modeling methods.

2.1.1 Markov Models

Markov models (also known as *Markov processes*) are considered one of the best modeling techniques for stochastic processes, in term of their mathematical tractability. Markov models are a compromise between the real-world dependencies involved among various physical phenomena and the theoretical requirement of "independent" phenomena to make the calculations possible [88]. The class of Markov processes that is often used for computer system modeling is called *Markov chain* processes. Markov chains are built by considering the stochastic process (e.g., parallel computation) consisting of a set of states that are visited once or repeatedly over time. This set of states is termed as the state-space of the system. A Markov chain process is defined as one whose transition to a future state from its present state depends only on the present state, and not the complete past history of its states up to the present state. Therefore, dependencies for predicting a future state are restricted to only the present state which is independent of other states. This type of dependency structure proves to be a very powerful tool for building up models of fairly complex real-world phenomena that can be solved.

There are three issues involved in modeling a computer system as Markov chain processes that are given by [100] as:

1. defining the process as a Markov chain according to the definition of a Markov chain given above, by analyzing the dependencies of various states of the process;
2. mapping computer system models to Markov processes; and
3. solving Markov processes.

Since a multicomputer system exhibits very dynamic behavior which strongly depends on contention for and sharing of system resources at a given time, a model of such a system relies on representing the internal states of the system. A detailed representation of internal states (i.e., dependencies among them) is sufficiently complex and leaves us without any methods of solution of the model other than simulations. This will happen when we want to analyze the states of the system in response to instruction and data streams of individual programs. The representation of current and past states provides a "memory" of current and past states of the system. If this memory includes very few details, then it will be difficult to predict the future states accurately. However, inclusion of a number of states makes the numerical solution of the model impractical. As mentioned above, Markov processes are considered a compromise between these two extremes. We can make up a Markov model of a given computer system as described (in words) in the following [100]:

> *Assume that we represent all possible states of the system by a set of mutually exclusive and collectively exhaustive states. Also assume that the future behavior of the system depends only on the current state of the system and is independent of previous states of the system. The times between corresponding entrances to and departures from ("holding times" or "transition times") are independent and identically (exponentially) distributed. Then the states of this system with a set of transition probabilities between these states correspond to a Markov chain process.*

After setting up this model, the next step is usually to represent the state transitions in a matrix form. This representation reduces various types of calculations regarding this system model to simple matrix manipulations [4, 88]. The transition probabilities represented by the matrix can also be represented graphically as shown in Figure 2-3. The circles show the states of the system during the execution of a program and p_{ij} are the probabilities of transition from state i to j. The solution of this model for more complex structures has also been an active area of research. [55] describe a technique of solving

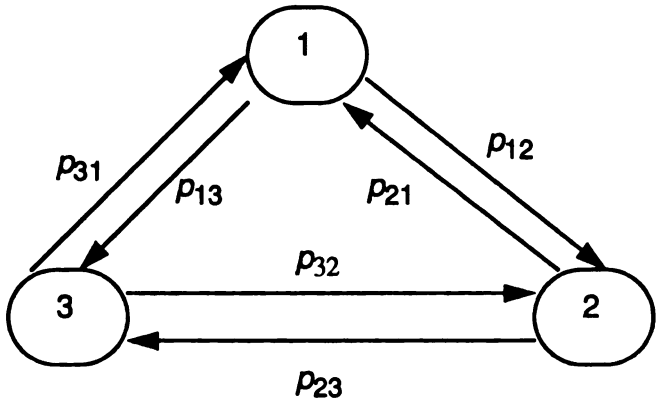


Figure 2-3. Markov Chain representation of the system-program behavior

these models utilizing fully symbolic and computer-algebra based methods at the earlier stages of Markovian process analysis and using numerical methods at the latter stages. [4] provides some examples of solving these models by purely numerical techniques, which work well for predicting future states of the system with simple models.

Markov models have been used to model various aspects of parallel and distributed computer system behavior and performance. [2] analyze the performance of the interprocessor communication architecture of the CM-2 by developing a discrete-time Markov chain model of its network architecture. The model shows the use of simplifying assumptions in a practical situation and predicted results compare favorably with the results revealed by a simulation study. [1] analyzes the time-dependent behavior of programs by using the program execution sequence to make up a homogeneous semi-Markov chain model. This model can predict the system performance with program parameters different than those used in the input program execution sequence.

2.1.2 Queuing Models

Queuing models are useful analytical tools for the analysis of systems in which conflicts develop when several entities try to access the same resource simultaneously, creating a loss of time for the system as a whole to perform a job. Although queuing models can be taken as a special case of Markov chain models, their wide-spread use to model multicomputer network communication and contention of resources among various jobs

makes them worth discussion independently from the Markov models. The study of queuing models is a discipline in its own right, known as *queuing theory.*

Queuing models are based on two abstractions: *servers* and *customers*. Servers are the resources of the system that have to be shared or utilized by the customers. A *queue* is another abstraction of this model where customers arrive and wait for the service, while the servers are busy to service other customers. This basic set up of this model is represented by Figure 2-4 which shows the servers by rectangles and customers by circles. A queue is shown by a line of customers with vertical bars on both sides. Queuing models consisting of more than one server are referred to as *queuing network* models. There are

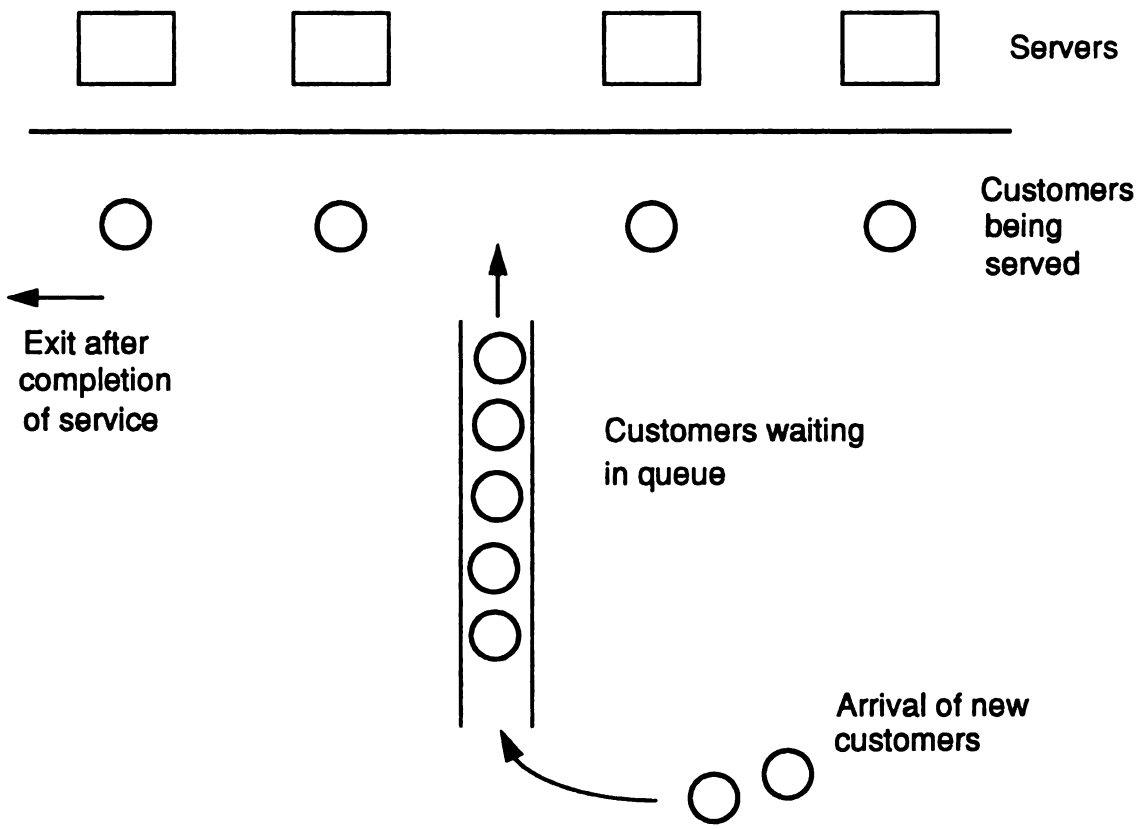


Figure 2-4. Ingredients of a basic queuing model.

three stochastic processes that have to be defined to parameterize this model. These processes (or random variables) are: (1) inter-arrival times of customers (input process),

(2) service time of each of the customers, and (3) number of customers waiting in the queue at a given time. The probabilistic definition of these parameters (i.e., their probability distributions) determines the queuing model for such a system. Once this model is determined, it can help answer the following questions:

1. What is the average waiting time for a customer in the queue?
2. What is the optimal number of servers needed for this system?
3. What about having a separate queue for each server? Will it be optimal compared to a single queue?

It can be appreciated that these questions are of a general nature and can arise in any phenomenon that is being analyzed by queuing models. The answers are very important to analyze the performance of a computer system (e.g., its throughput and response time). For modeling a computer system, often three types of devices are encountered [58]. These devices are:

- Devices that have a single server, whose service time does not depend on the number of jobs in the device. Such devices are called *fixed-capacity service centers*. For instance, CPUs in a system may be modeled as fixed-capacity service centers.
- Devices that do not exhibit "queuing" behavior, and jobs spend the same amount of time in the device regardless of the number of jobs in them. These devices can be modeled as a service center with infinite number of servers and are called *delay centers*. A group of dedicated terminals is usually modelled as a delay center.
- Devices whose service rates may depend on the load (i.e., number of jobs in the device) are called *load-dependent service centers*. An interconnecting network between the nodes of a parallel computer system is an example of a load-dependent service center.

It is common practice to consider the service times of all servers in a computer system as exponentially distributed [58, 69]. However, there are various cases listed by [58] where queuing network models are not very accurate.

[115] use a queuing networks to model a commercial multiprocessor bus. Important characteristics of the bus such as asynchronous memory write operations, in-order delivery of responses to processor read requests, priority scheduling of memory responses, upper bound on the number of outstanding processor requests, and so on, can be modeled accurately.

2.1.3 Petri Nets

A petri net is a graphical modeling tool for description and analysis of concurrency and synchronization in parallel systems. Petri nets were introduced by C. A. Petri in 1962 and are widely used to model asynchronous systems and concurrent processes. The theoretical problems associated with petri nets have been thoroughly investigated and therefore, it has sufficient mathematical structure to support formal analysis of parallel systems. The success of petri nets is mainly due to the simplicity of the model that works well to depict the complexity of large-scale systems. Many extensions have been added to the basic petri net model to facilitate their use for different application fields. These extensions include timed petri nets for quantitative performance analysis of systems, stochastic petri nets which uses random variables to specify the behavior of the model with time, as well as others [69]. Stochastic petri nets are considered more attractive as a modeling tool for analysis of multiprocessor systems.

The structure of a standard petri net is a graph that consists of *places*, a set of *transitions*, and a set of *directed arcs*. A place represents some conditions and a transition represents an event. Arcs connect places and transitions to each other. A place is an input to a transition if an arc exists from the place to the transition, and is an output of a transition if an arc exists from the transition to the place. Therefore, the set of arcs can be partitioned into a set of transition input arcs and a set of transition output arcs. Figure 2-5 is a graphical representation of an example petri net, taken from [69]. Circles (or nodes) represent places and bars represent transitions. Tokens are placed on place nodes to represent that a certain condition is held on that node. A transition bar (an event) can fire (occur) if all the nodes (conditions) that input to that transition bar have tokens (i.e., are holding conditions). When a transition bar fires, it removes one token from each of its connecting input nodes and places one token on each of its connecting output nodes [22]. This firing of a transition is called the *execution* of the petri net. For instance, if P_1 in Figure 2-5 contains a token, then transition t_1 will be enabled. States of a petri net can be determined by observing the collection of names of the nodes that hold tokens. The

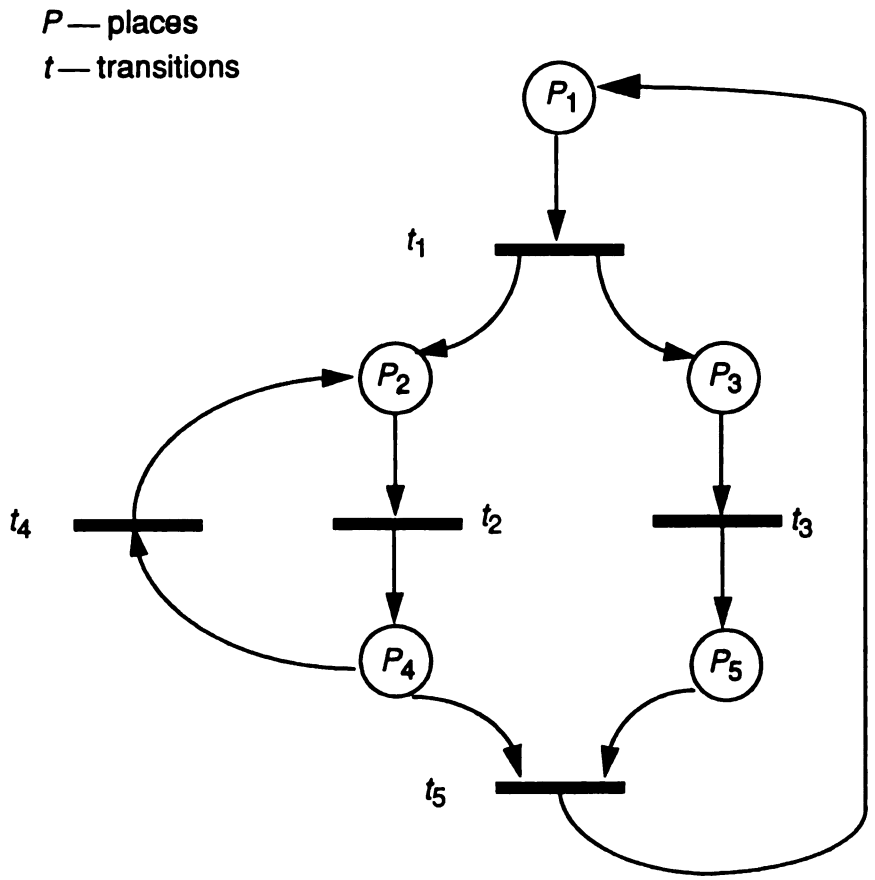


Figure 2-5. Example of a petri net.

number of tokens a node holds is equal to the number of instances of a node name in the state. This procedure is called *marking* of petri nets.

Petri nets and its variations are graphical, mathematical tools that can model the following characteristics of a concurrent computer system [22]:

- representation of concurrent execution of multiple processes;
- representation of nondeterministic and asynchronous executions;
- decomposition and composition into many graphs; and
- representation of model's structure and dynamic behavior.

Petri nets are also used to model the behavior of a processing unit that needs a resource to perform some action [69], the behavior of file systems [22], etc.

2.1.4 Simulation Modeling

Simulation models are used to model the operation of a system, rather than the structural components of the system [64]. Simulation is a very useful technique to solve an analytical model of a complex stochastic system which is very difficult to be solved numerically or symbolically. Usually, simulation is used in conjunction with analytical modeling of a system to verify the results. Simulation models have their own advantages, in addition to being a tool for verifying analytical models, that are discussed in [58, 63, 64]. Simulation allows much more detailed study of a system than analytical models. In this case, a more detailed model is considered a better model as it makes fewer assumptions. Simulation studies are useful for the analysis of systems even after they are realized because these models do not perturb the system behavior for analyzing it. There are several types of simulations used for computer system performance analysis. These include emulation, Monte Carlo simulation, trace-driven simulation, and discrete-event simulation. Simulation models are translated into simulation programs that generate statistical information regarding the model. This simulation data is analyzed by various statistical techniques [58].

2.1.5 Analytical Modeling Tools

In this section, we briefly describe some typical analytical modeling tools to analyze parallel system performance. The description of these tools will be within the scope of theoretical details of analytical models discussed so far.

Abrams, Doraswamy, and Mathur describe a tool called Chitra that produces a semi-Markov chain model of the program execution sequence (PES) which is observed during the execution of a program [1]. The design of Chitra has two aspects: (1) building up a semi-Markov chain model to represent the program behavior, and (2) using methods often applied in signal processing and in post-processing of simulation output, in addition to conventional visualization methods. Program behavior is modeled by considering various states of a program (state-space of Markov chain). Transitions between these states might occur due to conditional branches in the program. A semi-Markov chain process models

general state occupancy time distributions, which is considered appropriate for the software. The information obtained from the PES is used to determine the transition probabilities among different program states.

Chang et al. [22] have developed an analytical modeling tool-set consisting of a modeling tool TPQN, a textual specification language TPQL, and a simulator TPQS. This modeling tool integrates two popular analytical modeling approaches for parallel computer systems — timed petri nets and queuing networks— to model real-time complex systems. Existing modeling methods can model queuing behavior by queuing networks or asynchronous behavior of concurrent processes by timed petri nets. These methods are inadequate to model complex systems (including multiprocessor operating systems and distributed systems) if used individually. Complex systems require the use of an integrated model consisting of both modeling approaches. This modeling approach has been applied to real-time multiprocessing systems. The model is defined by the user using a simulation language TPQL that provides textual specification of graphical entities of TPQN model. TPQL is used as an input to the simulations system TPQS.

Funka-Lea et al. [40] describe an analytical modeling tool called Q+, developed by AT&T Bell Laboratories. Q+ is an interactive graphical tool for performance modeling that allows the user to specify the model graphically and also represents the simulation output graphically. It is made up of six components: the graphics editor, text editor, Monte Carlo simulator, language interpreter, browser and utility set. A user can specify the system as a queuing network that consists of two entities: nodes that are graphical sites in the network and transactions (jobs, customers, etc.) circulating among the nodes. A node has a queue associated with it which can hold transactions. A queue's capacity can be positive and finite or infinite. These model components are placed and connected together by the graphical editor. The model is parameterized by using the text editor. Q+ uses Monte Carlo simulation to analyze the queuing model of the system. Q+ allows restructuring of the models by using calls to a C library interface. Q+ supports a hierarchical approach of system modeling by allowing subnetworks to be defined by the user that can be

incorporated in other models. Q+ has been used to model many systems ranging from large national telecommunication networks to microprocessor internals. Model sizes have ranged from one to several thousand nodes. Perhaps the greatest advantage of Q+ is that it allows the system designers to make up their own model without requiring any expertise to be able to develop analytical models. Therefore, system designers can simulate the model to decide on alternative design options without having to explain the system to a modeling specialist to create a model for the system. Q+ also provides interfaces to two analytical packages to solve queuing networks.

2.2 Monitoring-Based Methods

Monitoring-based methods are effective for performance analysis of programs on existing systems. Behavior of a parallel computer system is monitored by observing various system and/or program parameters during the execution of a program through hardware or software means. These methods have been developed to provide the user some feedback about the behavior of a program when executed on a parallel computer system.

The objectives of analysis of an existing parallel system are usually different from those of a system in various design stages. Design options for system hardware and interconnecting network topology have already been made, and the performance of the system is dependent heavily on the programs that are executed on this system. Problems solved on a parallel system are expected to achieve significant speed-ups to justify the cost of the hardware as well as the development of a parallel program for a given problem. Speed-up is not enough to represent all aspects of the performance analysis because performance of a typical parallel system is a complex, multidimensional assessment that depends on various dynamic aspects of the system hardware as well as software, which might not have been specified prior to the execution of the program. From a user's perspective, the following questions can be asked about the behavior of a program executed on a parallel system:

1. Is the program running correctly?,

2. How well is it utilizing the available resources of the system to get maximum benefit of the concurrent processing?, and
3. How can the performance of the program can be improved upon?

The first question is concerned with the issues related with program debugging, while the second and third questions are typically related with program performance evaluation and optimization. Casavant [20] describes two of the most "ominous" characteristics of parallel programs that make them difficult to understand by the traditional techniques by parallel algorithm specialists, formal language experts, and operating system researchers. These characteristics are *topological mapping* and *synchronization.* The topological mapping issue demands that the spatial relationships between program data and processors should be understood. The synchronization issue demands correlating the events occurring at different times within a processor ensemble. When a program using even only a few processors is monitored, it typically generates a huge volume of data that is not possible to be assimilated by human senses. The idea of program *visualization* was introduced to attack this problem by graphically representing the numerical data. Visualization exploits the visual perception capabilities of a human user that are much more powerful compared to the textual perception capabilities [116, 117]. Several program visualization tools such as Seecube, GIST, ParaGraph, SHMAP, etc. are considered pioneers in this field [23, 27, 50].

The use of program monitoring with visualization for the purpose of performance analysis and optimization can be illustrated as shown in Figure 2-6. The process starts with the programmer developing a program to solve a problem on a parallel system. This program is instrumented usually by instrumentation libraries linked with the program. When this program is executed, it generates the observed information about the occurrence of various events during the execution, in addition to the required results from the program. The observed trace data is typically a multi-dimensional dataset which is analyzed by using various displays of the available visualization tool to represent several perspectives of the program behavior. The programmer has to develop a "mental bridge" [95] to link the behavior and performance of the program as represented by visualization displays to

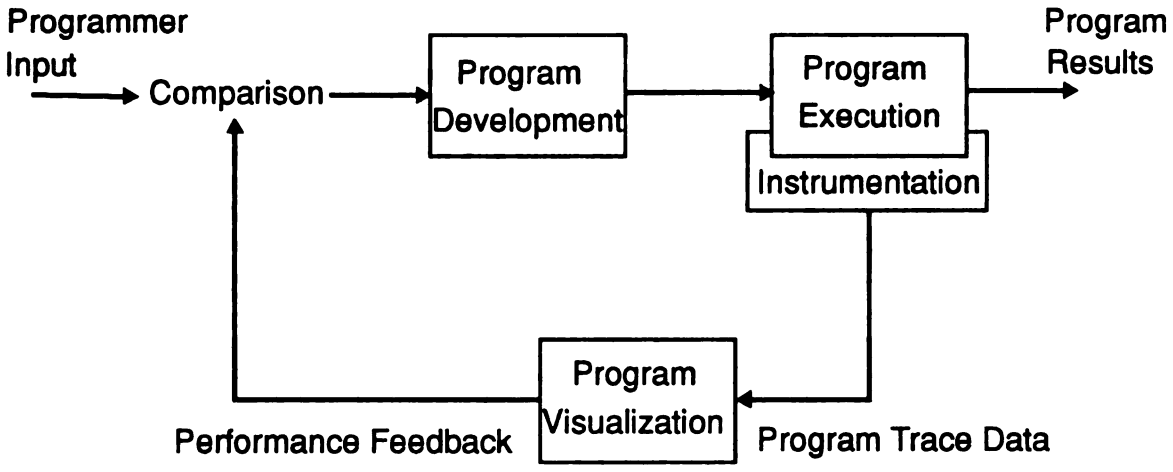


Figure 2-6. Monitoring-based methods for performance analysis.

the actual program. This mental bridge is necessary to establish for debugging the program for correctness as well as optimizing its performance.

The program monitoring approach is not as general and mathematically rigorous as the analytical modeling approach. This lack of generality and rigor has its advantages as well as disadvantages. It is beneficial for the purpose of performance analysis because:

- It is the most direct approach to analyze a particular program on a real particular parallel system without having to make any simplifying assumptions about the behavior of either of them.
- Behavior of various system and program related parameters such as processor utilization, message traffic on various communication channels, memory references, cache hits or misses, lengths of system queues, routing protocols, etc. can be observed which contributes directly to the programmer's understanding of the program activities during its execution.
- A programmer does not have to build a model for a program which requires specialized model-building expertise.
- The cost of instrumenting the program in terms of the programer's time needed to instrument it is usually insignificant.

The program monitoring approach is not free from limitations. Its major limitations include:

- Lack of rigor in the approach typically manifests itself as lack of scalability of monitoring-based approaches for large-scale parallel systems or large volumes of trace data for large number of events.
- Lack of extensibility, due to which it is difficult to perform analysis which is not supported or provided by a visualization tool. Often trace data semantics are such that the data can not easily be used with any other analysis tool.
- An analytical approach can help decide an optimal design alternative because different options can be compared against the same model. This type of comparison among various parallel implementations of an algorithm that can help the programmer to improve the performance is usually non-existent in monitoring-based program visualization tools.
- This is an intrusive approach and perturbs the normal execution of the program.

As mentioned in Chapter 1, monitoring-based methods were introduce to cater to the needs of the users that could not benefit adequately from existing analytical techniques. The limitations of the monitoring-based methods have resulted in improving the approach behind these methods. For example, researchers are developing new representation and analysis paradigms. To put these efforts in an appropriate perspective, this section has been divided into two parts: conventional and advanced. Program monitoring tools will be discussed with the appropriate analysis approaches used by them because almost every tool represents the realization of a specific approach. Therefore, these approaches have been categorized as conventional and advanced, with certain aspects discussed in the following sub-sections. Also, performance data instrumentation techniques will not be discussed as instrumentation systems are beyond the scope of this thesis. The reader is referred to [103, 109] for more details.

2.2.1 Conventional Methods

We refer to the performance analysis methods and tools that do not process, analyze and present (visualize) the performance data by rigorous means as conventional methods. Most of the visualization tools that replay the program execution based on the execution trace data will fall in this category. Information is presented by typical program visualization displays with some single-value metrics, such as average system utilization, the cumulative busy or idle time of each processor, the number of messages sent or

received, etc. Typical graphical displays include: time-series display (strip chart), space-time diagram, Gantt chart, Kiviat diagram, bar graph, dial, LED display, x-y plots, matrix display, pie chart, contour plot, three dimensional scatter plot, program call graph, architecture topology graph, meter, vector plot, etc. [103]. These are exemplified in specific tools that we will discuss here. There are three common points among all the above typical visualization displays and hence in the tools that use them:

1. These displays show temporal and/or spatial activity of the program on a parallel system.
2. The displays are driven (or updated at each step) by the event information obtained from the trace data that is ordered according to the event occurrence times, by applying some rule.
3. The learning time for these tools is relatively small as the displays tend to be intuitive.

The rule for updating a display is typically not rigorous and depends heavily on the data semantics defined by the instrumentation system. Analyzing the graphical displays and relating them back to the program have to be done "manually" by the user. This approach might seem very rudimentary, but according to several surveys providing user feedback about these tools [23, 61, 77, 92], the number of users using these tools as a part of their program development environment is much larger than those using performance analysis tools of any other type. This is probably due to their simplicity. Therefore, despite their limitations, these tools can not be overlooked because various advanced tools (discussed in section 2.2.2) have been developed based on the experiences with these tools.

Conventional analysis methods are mainly of the following types: visual, query-based, debugger-based, heuristic-based, statistical and textual. Conventional analysis methods and tools will be discussed according to this classification.

Visual analysis involves graphical displays of information and the ability of the user to visually perceive and discern features, trends, patterns, etc. in the displays. ParaGraph can be considered a representative tool belonging to this category [50]. There are very few program visualization tools that can match the variety of displays provided by ParaGraph. It uses the trace data obtained by instrumenting programs using the Portable Instrumented

Communication Library (PICL) [44]. Several of its displays that contribute to visual analysis include processor count, Gantt chart, space-time diagram, concurrency profile, utilization meter, Kiviat diagram, message queues, and task graphs. The SHMAP (Shared Memory Access Patterns) tool is designed to aid in the study of memory access patterns, cache strategies, and processor assignments for matrix algorithms in shared-memory systems. There are two main displays—one to show loads from the main memory and the other to show stores to the main memory. Each memory access results in illumination of the corresponding memory element.

A number of general approaches to envisioning information are described by Tufte [116, 117]; one is "analysis by small multiples" for detecting changes between views (which we routinely apply without naming it per se). Another approach that is routinely used is "analysis via multiple views," which is more formally stated in [65], whereby multiple views assist us in navigating through the space of processor/process states, interactions, and time. Another notable example of visual analysis is "projection pursuit," whereby the user refines hypotheses about performance by following a path of selective projections of multi-dimensional data [28].

Query-based analysis involves techniques typical of database or spreadsheet processing to extract information from the performance data. The data is viewed as some form of a relational database and can be selectively accessed via queries [105]. "What if" questions can also be supported to enable prediction [41]. SIEVE is a program debugging environment [41] that treats the trace data as a temporal, relational database. The structure of SIEVE allows the user to access the source code structure and event histories as a relational database.

In *debugger-based analysis*, an interactive debugger supports performance analysis by coupling source-code-level information with performance visualizations. Performance is then more closely related to the programmer's viewpoint, and visualizations may be dependent on the programming model. MPPE (MasPar Programming Environment) is an integrated graphical debugger, performance profiler, and visualizer, with client-server

architecture for remote debugging [23, 70]. It allows source-level debugging and static analysis of parallel programs on the MasPar family of data-parallel computers. PRISM is another example of a source-code level debugger and graphical performance analysis tool used with the CM-5. It uses regular field visualization techniques to examine large arrays [23].

Heuristic-based analysis may be done automatically or interactively (with user input) based on static and/or dynamic information. It is applied, for example, to optimize the mapping of the program or the re-structuring of code (for parallelization). These methods are also used to detect non-determinacy due to the asynchronous nature of the processes and race conditions due to shared variables in parallel programs. Callaham and Subhlok describe the PTOOL tool [16] for detecting and analyzing asynchronous parallel loops in a program.

Statistical analysis involves calculations on the basic metrics to derive (single value) characterizations such as mean, median, skew, standard deviation, maxima, minima, frequency distribution, etc. Statistical calculations are common in many tools. Tools such as ParaGraph [50] and Seeplex [28] display statistical information in addition to the graphical information.

Finally, *textual analysis* is the brute-force inspection of program or trace data. This method has to be used when the performance visualization tool does not adequately provide the information that a programmer needs to know either to debug it, or to understand its behavior and performance bottlenecks. Various tools such as ParaGraph recognize this fact and provide a means to browse through the trace data.

Thus, parallel program performance analysis in support of verification, optimization, and prediction is an active area that has not yet settled upon definitive methodologies, especially for large-scale systems. It will benefit from continued basic and experimental research, which is the motivation for this work.

2.2.2 Advanced Methods

Performance visualization and analysis methods and tools that try to overcome the limitations of the conventional tools have been categorized as advanced methods. These methods usually develop their approach on the basis of various well-known rigorous analysis methodologies often used in disciplines such as statistical analysis, numerical analysis, and signal processing. Some of the tools do not use any rigorous approach but their approach alleviates certain limitations of the tools described as conventional tools in section 2.2.1 are also discussed in this section.

Advanced methods of analysis include: pattern analysis methods, data transformation methods, numerical methods, automatic analysis, hierarchical analysis methods and auralization. These methods and tools using them are discussed next.

Pattern analysis methods are perhaps among the first techniques that were explored upon recognizing the limited effectiveness of conventional methods. [89] suggests that analysis of patterns of programs should be a potentially useful technique for performance analysis and debugging. Belvedere [51] is perhaps the only debugger that was designed to explicitly analyze the patterns. Hough and Cuny [52] indicate the fact that patterns generated by communication calls in a parallel program can help identify communication related bugs in a program. "Perspective views" are supported by the Belvedere tool, which depict abstract events in a space-time diagram (via a reordering transformation) so as to emphasize behavior patterns from the programmer's viewpoint [51].

Data transformation methods applied on program trace data to visualize it from various perspectives can be considered as advanced methods. The development of tools based on these methods give the user (or analyst) freedom to deal with any type of performance data. The user has to choose an appropriate transformation technique (according to the rules set by the tool) to select a particular dimension of the data for visualization. Pablo is a performance evaluation environment that uses this approach [86]. It defines a standard data description format (SDDF) for representation of the trace data which makes it easier to share the performance data among multiple tools. Pablo uses a data-flow model of

visualization where the user to manipulates and interconnects various data transformation modules in the form of an acyclic graph to transform the data and display it. Analysis methods discussed in this thesis are also based on transformation of the performance data. However, we model the performance data explicitly as a matrix and then apply transformations to this matrix in a rigorous fashion. In addition to transformation of the data to lower-dimensional data, we transform this lower-dimensional data into other forms for analysis.

Numerical methods have been applied on performance data to obtain information about specific aspects of the program. Such methods are handy when the performance data is already in a form on which such rigorous analysis could be carried out. [17] has shown the analysis of time-series plots of processor utilization by smoothing, rounding, and piecewise polynomial fitting techniques in order to identify "phases" of the program and predict their scalability characteristics.

Automatic analysis techniques are concerned with providing program-specific information without explicit user input about the program. These techniques rely completely on low-level performance data to extract the knowledge of high-level abstractions of the program. These methods do not require explicit user input because the performance data that they use is already in a form that can be processed by standard data analysis methods that are often used in statistical analysis and signal processing. IPS-2 [73] supports analysis of phases by applying smoothing, segmentation, and combining techniques on a curve representing a performance metric. However, in certain cases, this type of automatic analysis might not be as effective as the analysis performed with some amount of user input [74].

Hierarchical analysis methods have been proposed to analyze the performance of programs executed on large-scale parallel systems [29]. Users can hierarchically navigate through views in a global context showing only the macroscopic information and can selectively see parts of it in more detail. Processors can be categorized according to their

behavior, and statistics corresponding to that category can be displayed. The execution visualization tool Seeplex [28] implements this type of categorical views and is scalable.

Auralization, that is, representing the program behavior as sound, is a comparatively new addition to the performance visualization area. Program auralization can be considered as a complement to visualization using the sound capabilities, in addition to the high-quality graphics capabilities, that are available in today's workstations. Auralization is also another example of transforming the performance data from its original form (or domain) to another form which contributes towards the efficacy of the analysis of the original problem. [39] uses a prototype sound tool to represent the message-passing behavior of the program which can be used with a conventional visualization tool to enhance the user's understanding of the behavior of the program. Pablo [86] has also adopted the idea of sound in addition to graphics to represent the program activity.

Advanced methods and tools have attempted to address the problems and limitations of conventional tools. Most of the tools mentioned in this section are still in various stages of development. While advanced methods have advantages over conventional tools, they are typically not easy to apply. Usually a user faces a very steep learning curve to use a new more sophisticated tool. Therefore, the user-base of these tools is expected to be smaller compared to that of conventional tools. This factor offsets the advantages of these tools to some extent, because the methods were developed to be used! The practical benefit of these methods and tools, therefore, remains essentially unknown unless there is wider usage and user feedback becomes available.

2.3 Application-Specific Methods

Application-specific methods of performance analysis and debugging can be considered as a version of program monitoring that is free from any restrictions of generality of approach and re-usability of displays. These methods can be considered as a reaction to the lack of information given by performance visualization methods and tools which provide the user with a direct understanding of a program. Application-specific

visualizations enable the user to create the displays that directly and most effectively show the behavior of the program. Performance visualization presents the behavior of a program in an indirect manner by representing various performance metrics graphically. The user must develop a link between the performance of the program and the program-specific information such as program data accesses and data values. Application-specific methods, on the other hand, use a direct approach to this problem and use displays that represent a particular program and its data values.

Figure 2-7 shows the process of creation of an application-specific visualization with an application program. A programmer develops a display in addition to the program. The

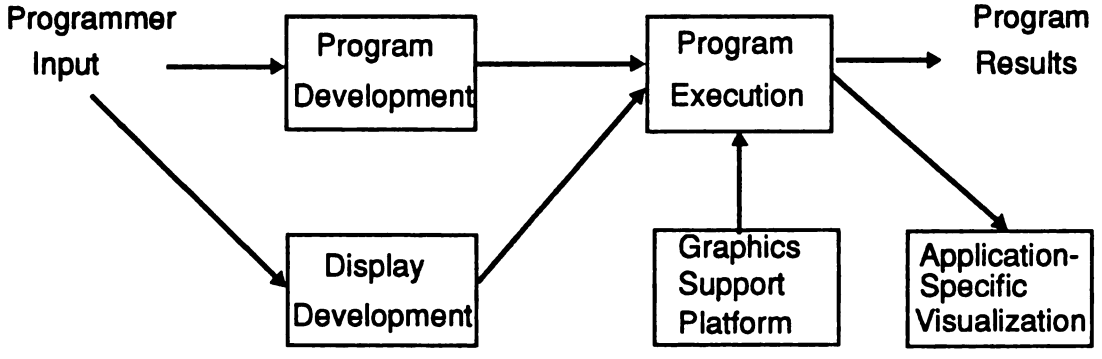


Figure 2-7. Application-specific methods for performance analysis.

display is dynamically updated by using available graphics libraries of the system where the actual program is executed. Both program code and visualization code are executed by the same system with the display code using the graphics library support. This process generates the actual program output as well as updates the visualization display. Therefore, visualization is a part of the actual program and can represent the program data.

We have categorized application-specific visualization methods into two types: algorithm animation and application-domain visualization. The only difference is in the way various tools have used them and is discussed in the following.

2.3.1 Animation

Animation has a special significance in computer graphics and visualization. Animation is used to show the passage of time by updating a display according to the change during an incremental instant of time. Animation has been used by generic performance visualization tools (discussed in section 2.2) in addition to the application-specific tools. For example, ParaGraph uses animation displays to show the state of the processors and their message-passing activities. ParaGraph also uses animation in application-specific displays that must be programmed by the user, such as to present a program data object and its values. However, these animations (known as algorithm animation) are not re-usable, i.e., the animations have special features that are appropriate for a particular program only and might not be used with any other. Zeus is an algorithm animation system that uses multiple views to animate various aspects of a particular algorithm [13]. Commonly used algorithms such as Quick Sort, multilevel adaptive hashing, topological sweeplines, and others have been shown effectively by using animation and sound. Zeus has been used for performance analysis and algorithm tuning purposes.

2.3.2 Application-Domain Visualizations

Application-domain visualizations are also program dependent and not re-usable. In contrast to algorithm animation, these visualizations are not used for performance analysis, but exclusively for debugging and correctness checking of a program. These displays are developed alongwith the application program to be visualized, therefore, they represent the application domain directly, in addition to its semantics and fundamental behavior. POLKA (Parallel program-focused Object-oriented Low Key Animation) is an application-specific visualization system [109]. It uses high-level graphical objects and

motion primitives. A POLKA animator object is to be instantiated in the program for developing the animation.

Although application-specific visualizations have been used successfully by various researchers, it remains an obscure art. Once developed, these visualizations are easy to use by a user, but it is hard to develop a good application-specific visualization display that can portray appropriate aspects of the program activity. Moreover, a user must be good at writing the application program as well as developing a powerful visualization in order to benefit from this approach. The user feedback on these tools is even lesser than that on advanced visualization methods and tools or analytical tools.

This concludes the survey of related work on the performance evaluation of parallel systems. It is clear that all three approaches discussed here have distinct advantages, yet none of them are free from constraints. We believe that the approaches which are based on mathematically rigorous and conceptually precise foundations are more likely to be effective, but to do so, must be implemented with a tool that caters to the needs and capabilities of its expected user. Starting in the next chapter, we will introduce a methodology of modeling the performance data and transforming it to other forms (or domains) where more effective analysis methods can be applied, and discuss the analysis methods in these multiple domains.

Chapter 3

Performance Metrics and Trace Data Modeling

As discussed in Chapter 1 and 2, there are various techniques to evaluate the performance of a parallel program executed on a parallel computer system. Two types of techniques are analytical modeling and monitoring [89]. Analytical modeling is concerned with modeling the behavior of a program on a particular parallel architecture by developing various simplifying assumptions to devise a feasible model of the system and program. Program monitoring (hardware or software) is concerned with logging the run-time data from the system or program. Program monitoring is considered a more effective technique than analytical modeling for complex systems because the actual behavior of the program can be observed and analyzed without any simplifying assumptions. Program monitoring is also used to verify the results obtained by analytical modeling techniques. Therefore, researchers working in diverse areas of parallel program development, modeling and analysis, and optimization have to rely on run-time trace data. In a practical situation, however, it might not be possible for these researchers to share the same trace data without appropriately converting it to a suitable structure. In spite of this, trace data is the only common ground for everyone working in the area of performance evaluation. Therefore, the analysis methods and tools developed in one area could be useful in others if a common trace data structure could be formulated. In this chapter, we will specify the performance metrics and trace data with the ultimate objective of this type of standardization at a "functional level".

3.1 Performance Metrics and Preliminary Definitions

The performance analysis of a concurrent system is an intricate problem to deal with, both at quantitative and qualitative levels. It is difficult to decide upon a set of generic enough objectives to analyze the performance of a parallel program (or system) because the criteria of desirable performance might be different in different practical situations. A

program can be evaluated in terms of its characteristics of accuracy, reliability, understandability, testability, expandability, design cost, running cost, etc. [33]. All of the above characteristics are non-numerical descriptive variables. Therefore, they are not suitable for the analysis. Hence, the performance analysis tools have adopted the convention of characterizing the performance of a program (or system) by using various suitable numerical parameters. These parameters give some measure of specific aspects of the performance and are referred to as *performance metrics*.

Performance metrics change their values with respect to two independent variables: space and time. These are the two basic variables that specify the interaction of a parallel program with a parallel system. All other events and performance metrics are defined in terms of these two independent variables. For all the definitions considered in this chapter and elsewhere in this thesis, we have assumed that there are P processors being used for executing a program which is traced for N clock cycles.

Definition (Time)

Execution *time* of a parallel program is a discrete integer, denoted by n such that $n \in \{0,..., N-1\}$, where $n = 0$ specifies the instant of time when tracing of the program began and $n = N-1$ specifies the time when tracing was stopped. ◆

The range of time variable $n = 0,1,..., N-1$ defines the global time scale for the program executing on a given system. All other events and hence the performance metrics are defined with reference to this time scale. We will treat the time scale (or index) as a vector quantity, such that:

$$n^T = [0\ 1\ -\ -\ -\ N\text{-}1] \tag{3.1}$$

where the $n = 0$ is arbitrarily chosen by the programmer to selectively start instrumenting a part of the whole code. The length of the time scale (N) is determined by the time taken to execute the instrumented portion of the code.

Definition (Space)

Space of the program (or system) is specified by a discrete number, denoted by p such that $p \in \{0,\ldots\ldots, P-1\}$, where P is the total number of processing nodes allocated for the program, where a node includes a processor, local memory, routers and other components necessary for localized computation in a distributed system. ◆

The space of the system is defined by the logical processor number assigned to each of the processors. The events in a distributed system are identified in terms of the spatial locations of their occurrences in addition to the global time-stamps associated with them.

Definition (Performance Metric)

A *performance metric* is a discrete number specifying a certain aspect of the behavior of a program and/or its interaction with the system by using numerical values. A performance metric will be denoted by m. ◆

All performance metrics are functions of the time and space variables defined above. There are several types of performance metrics, such as response time, processor utilization, processor state, operation count, computational load, execution rate (throughput), message count, message volume, communication time, communication overhead, I/O traffic, etc. [89]. Generally, these metrics are sensitive to the occurrence of specific types of events, usually identified by an event-identifier. A trace data log contains some event-specific information in addition to the temporal and spatial location of an event. This information is used to update the values of affected metrics at a given temporal and spatial location. Performance metrics are useful to define the *system states* and the *dynamic behavior* of the system. Frequently, "system state" will be used to represent the value(s) of a performance metric at a particular instant of time. Since the behavior of the system evolves over (execution) time, we need to define the *dynamic behavior* of the parallel system.

Definition (System State)

State of a parallel computer system while executing a program is a numerical representation of the activity of each of the processors of the system at a discrete instant of execution time, where the activities of a processor might include computation,

communication, I/O etc. The numerical values of system states can vary with time due to the occurrence of specific events. ◆

In the context of this work, the activities of a processor in the system will be restricted to two broad categories: (1) computation and (2) communication. A processor can be considered "busy computing" if it is not participating in any message-passing activity, according to the definition used by [45]. The dynamic (instantaneous) system state will be represented as a vector that will be called a *state vector*, defined in the following.

Definition (State Vector)

A *state vector* denoted by $v(n)$ consists of P elements denoted by $v_0, v_1, ..., v_{P-1}$ that hold the states of each of the processors $0,1,..., P-1$, respectively, at each instant of program execution time n. ◆

The state vector is initialized to an appropriate value at the start of the tracing, depending on the nature of the selected performance metric. This vector will play a key role for the transformation techniques presented in subsequent chapters. We can numerically denote the busy state of a processor by 1 and the idle state by 0. Whenever we will need to reduce the state of the whole system at a particular time to represent it as a single numerical value, we will use the common reduction technique of determining the ratio of the processors that are busy computing to all the processors allocated for the program, by utilizing the dynamic information about the state of the entire system held by $v(n)$. In some cases (especially for spatial domain analysis as discussed in Chapter 5), we will use cumulative system states as well where we do not need to reduce the whole system state to a single numerical value, but reduce the states of each processor as a ratio of their busy time to the total execution time up to the observation instant.

Definition (Dynamic Behavior of the System)

Dynamic behavior of a concurrent computer system during the execution of a program is represented by some performance metric that varies with time as well as with space. The observation of dynamic behavior generates time-series data. ◆

Dynamic behavior of the system will be characterized by system states at each instant of time (or with each occurrence of events of interest) during the execution of a program. We

will need to reduce the state of the whole system to a single numerical value due to the characterization of the dynamic behavior that we will define and use in subsequent chapters.

3.2 Execution Trace Data and Matrix Representation

Performance visualization and analysis tools use the time and space as independent variables to depict the occurrences of various types of events during the execution of a program. Currently, there is no standard format of representing the trace data. Every instrumentation and visualization tool sets its own data structure and semantics. If generalized performance analysis methods are proposed, then data semantics will have to be resolved to make them work. This fact points to the need of having some generic data representation format that could be helpful for performance analysis at the functional level, i.e., the analysis methods must be independent of a particular tool's data semantics. This idea is known to have worked in several other disciplines and applications. For instance, in computer graphics, the data is handled in terms of vectors and matrices to represent various geometric shapes and functions. However, it is not yet practical to have a common data logging and representation strategy for all instrumentation systems. So, a practical alternative is to have a preprocessing filter (a program) to convert the trace data to a format suitable for analysis. We propose a matrix representation as the general performance data structure. This convention has been adopted from statistical analysis techniques that use a "primary data matrix" to represent the data [26]. This primary matrix is transformed to other forms for better analysis. We have adopted the same terminology and create the primary matrix by preprocessing the "raw" trace data. The primary matrix is then transformed to other domains (or subspaces).

Performance data in its raw form is ordered according to the time-stamps of events of interest. At each time-stamp, there might be several events occurring at various processors concurrently. Such events are distinguished by their processor number and event-identifier. For the purpose of generality, we represent this data by a *trace data matrix T*, although we are not going to use this matrix for subsequent analysis.

Definition (Trace Data Matrix)

Trace data matrix is represented by a block matrix given as:

$$T = \begin{bmatrix} E\langle n_0, p_i\rangle \\ E\langle n_1, p_i\rangle \\ \cdots \\ \cdots \\ E\langle n_{N-1}, p_i\rangle \end{bmatrix} \tag{3.2}$$

where $E(n_0,p_i)$ are the block matrices showing concurrent events at an observation instant n_0 and at processor p_i, where $i \in \{0, 1,..., P-1\}$ and are given by:

$$E\langle n_o, p_i\rangle = \begin{bmatrix} e_0\ n_o\ p_i\ C_0 \\ e_1\ n_o\ p_i\ C_1 \\ \cdots\ \cdots\ \cdots\ \cdots \\ \cdots\ \cdots\ \cdots\ \cdots \\ e_k\ n_o\ p_i\ C_k \end{bmatrix} \tag{3.3}$$

where $e_0,e_1,...,e_k$ are the event identifiers of $k+1$ concurrent events that occurred at $k+1$ (out of P) processors at some observation instant (time-stamp) n_0. Blocks $C_0,C_1,...,C_k$ represent the event-specific information where each of these blocks consists of one row and the number of columns determined by the event type having maximum number of event-specific fields of information. Therefore, some event types will have redundant columns to make the matrix notation possible. ◆

The trace data matrix T includes event information obtained from instrumenting the program and ordering it according to global time stamps. It contains more information than what would be necessary to carry out analysis with specific well-defined objectives. Hence, preprocessing is done on this matrix.

3.3 Metric-Based Representation

A better and more useful representation of the trace data is in terms of a performance metric of interest. This is essential to reduce the dimensions of the data to facilitate its analysis and visualization. We represent this data by a matrix M, called the "primary metric-based performance data matrix".

Definition (Metric-Based Performance Data Matrix)

Metric-based performance data matrix is represented in the block form as:

$$M = \begin{bmatrix} M\langle n_0, p_j\rangle \\ M\langle n_1, p_j\rangle \\ \cdots \\ \cdots \\ M\langle n_{N-1}, p_j\rangle \end{bmatrix} \tag{3.4}$$

such that:

$$M\langle n_i, p_j\rangle = \left[n_i\ p_j\ m(n_i,p_j)\right] \tag{3.5}$$

where $i \in \{0,1,...,N-1\}$, $j \in \{0,1,...,P-1\}$ and $m(n_i,p_j)$ is the numerical value of the performance metric of interest at a particular time and spatial location, with the constraint that $n_0<n_1<...<n_{N-1}$. ◆

There is a row in this matrix for each occurrence of an event that affects the metric's value. The matrix has three columns, therefore, it reduces the trace data to a lower-dimensional subspace. The first column lists the time-stamp of the event associated with that row (in ascending order). The second column lists the processor number. The third column lists the corresponding value of the metric. There might be several rows having the same time-stamp to account for concurrent events.

Lemma 3.1

The state of the system represented by matrix M remains unchanged between the two time instants indicated by the first column of any two successive rows.

Proof:

By definition of metric-based performance matrix M, each row indicates the occurrence of a relevant event that was logged by the instrumentation system. A "relevant" event is one that affects the metric's value. Let n_i be the time-stamp (first column) of an event represented by ith row of M and n_{i+1} be the time-stamp of the event represented by the i+1st row. Since the instrumentation is event-driven as specified earlier, there can not be any other relevant event during the time interval $n \in (n_i,n_{i+1})$ except for the two events represented by ith and i+1st rows. The state of the system under event-driven tracing does not change unless an event occurs that could change the state. Since there was no relevant event during the interval (n_i,n_{i+1}), this implies that there was no event that

could have changed the state of a processor p during this interval. Hence, the system state remains fixed at its value at time-stamp n_i, at least up to the time-stamp of the immediate next relevant event that occurs at time-stamp n_{i+1} (listed in the i+1st row) when it might then change. QED. ■

This lemma will be used throughout the processing of the metric-based performance data matrix M, in order to determine the system states and dynamic behavior.

Theorem 3.2

The metric-based performance matrix M can be used to specify the concurrent computer system states and dynamic system behavior during the execution of a program.

Proof:

Suppose n_o is an observation instant such that $n_o \in [0,n_{N-1}]$, where the state of the system is to be determined and $m(n,p)$ is an appropriately selected performance metric indicating the state of a processor p at a time n. Suppose that there are P processors of the system being used for a computation, and a state vector $v(n)$ consisting of elements $v_0, v_1,..., v_{P-1}$ is used to hold the state of the system at time n. Then by definition, the metric-based performance matrix M represents all the events relevant to metric m, sorted according to their times of occurrence and listing the times, spatial locations and the corresponding values of the metric m. Each row of M indicates the occurrence of an event relevant to metric m and there are no relevant events between any two events represented by two successive rows of matrix M. If matrix M is browsed from the first row and values of $m(n,p)$ update the corresponding elements of the state vector $v(n)$ for all $n \leq n_o$, then by definition of the system state, the vector $v(n_o)$ will represent the system state at time n_o.

Let $v_0, v_1,..., v_{P-1}$ be the elements of a state vector $v(n)$ holding the corresponding values of an appropriately selected metric $m(n,p)$ for each of the P spatial locations of the system. If v_i is an observation of the metric at processor i at an instant of time n during the execution of the program, then all such observations can be collected as a time-series for the processor i over all instants of time $n \in [0,n_{N-1}]$ to represent the dynamic behavior of the processor i. Dynamic behavior of the entire system can be represented by applying a suitable reduction technique to combine the values of all the elements of the state vector $v(n)$ into a single numerical value for each observation at time $n \in [0,n_{N-1}]$. Then, these observations will represent the dynamic behavior of the entire system. QED. ■

The representation of the trace data in the form of a matrix does not provide any magical solution to the performance analysis problem. However, when it is used with an appropriate representation of the system (such as the interconnection topology, the decomposition topology, etc.) and the program (such as the programming paradigm, data

distribution, etc.), it can definitely increase the effectiveness of the analysis. Generally, there are the following benefits of using the "matrix approach":

- It is beneficial in terms of representing the data in a strictly uniform format, without much concern for the data semantics (once specified). This technique is used by performance analysis tools such as Pablo [86], where a data description header is used to interpret various fields in the performance data obtained from a particular instrumentation system. In the case of a matrix representation, the type of each column needs to be specified, and possibly symbolically named. Then, regardless of the operations performed on the metric-based trace data matrix, the result will be another matrix (or vector), which can be used with any other commercially available data analysis tool. This feature is available in Pablo, with the limitation that the matrix data can be fed to other analysis modules, instead of exporting to other tools.
- Matrix approach ensures generality and rigor of the analysis algorithms developed for the performance evaluation. Performance data forms a multidimensional data set, and matrices are known to represent multidimensional spaces. The transformations applied to these matrices can be considered as transformations from one vector space (subspace) to another. Generally, any analysis algorithm that accepts the data in the form of a matrix can be applied in any subspace. A set of analysis modules that operates on matrices is all that is needed. This not only simplifies the design of an analysis tool but also provides the user with more flexibility to select the analysis method, in terms of its efficiency, error properties, etc.
- Matrix approach has potential for efficient implementation on a parallel or vector architecture. This might be trivial for the case of analyzing the performance of small-scale parallel systems but it is particularly useful when one has to work with large-scale parallel systems. Matrices and operations on them are already in the mainstream of parallel programming.

The metric-based performance data matrix, written in shortened notation $M(n,p)$, will be a reference point for the analysis to be presented in subsequent chapters. The subspace (or domain) represented by this matrix will be called the *performance data domain*. The choice of metric represented by m is arbitrary and does not influence the analysis methods. The performance data domain and other *analysis domains* will be defined in the next chapter alongwith the transformation techniques that will be applied.

Chapter 4

Transformations to Analysis Domains

It is a usual practice to transform observed statistical data to other forms. The efficacy of statistical analysis and the model-building process is enhanced in certain cases when the primary data matrix is transformed to other subspaces [26]. Performance data is a multidimensional data set and is to be analyzed using visualization methods, therefore, transformation of this data to other subspaces having fewer dimensions is desirable. This reduction in the dimensions of data is a standard technique used in scientific visualization [14]. This same idea has been adopted here by transforming the raw trace data to a single-metric-based data matrix M, and it will be applied further to this matrix by transformation to the analysis domains (subspaces). These domains include:

- Spatial domain,
- Temporal domain,
- Event domain, and
- Frequency domain.

Each of these domains is a vector subspace having fewer dimensions compared to M, and can be characterized either as a digital image (in the case of a two-dimensional subspace) or as a discrete-time signal (in the case of a one-dimensional subspace). This characterization not only ensures mathematical convenience for analysis in the transformed domains but also makes it possible to utilize efficient image and signal processing algorithms to meet specific analysis objectives in these domains. These algorithms ensure scalability and extensibility in addition to the rigor of the "multiple-domain" analysis approach.

In this chapter, characteristics of and transformation to each of the above domains are presented using the matrix-based model for the performance data that was developed in the previous chapter. In addition to the analysis domains, the performance data domain is

also examined to get some insight about its nature. We end this chapter by briefly considering the application domain in the context of this work.

4.1 Performance Data Domain

In Chapter 3, program performance data was modeled by a metric-based performance matrix M. M has three dimensions at each instant of time when a specific event occurs, at which time a new row is added to M. The three dimensions of each row vector are: time-stamp of the event n, spatial location (processor) of the occurrence of the event p, and the corresponding value m of the chosen performance metric in response to the occurrence of that event. Therefore, an occurrence of an event that can change the value of the selected performance metric adds a three-dimensional element (row vector) to the performance data domain. It can be imagined that a metric-based performance matrix M, corresponding to a program that generates a "moderate" number of events of interest, will result in an immense number of elements in the performance data domain.

For the purpose of transformation of the metric-based performance data matrix M (i.e., the performance data domain) to other lower dimensional analysis domains, we will need to transform a subset of the dimensions of M to other domains. In order to support the transformations, we will need to count the events of interest listed in M, from the start of the tracing to a specified instant of time. The number of events of interest up to a given observation instant n_o is equal to the number of rows of M up to the time n_o. We define a time-varying counter for this purpose, which will be termed as *event counter* and denoted by $e(n)$.

Definition (Event Counter)

An event counter at an observation instant n_o during the execution time is denoted by $e(n_o)$ and holds the number of rows in the metric-based performance matrix M up to the time $n \leq n_o$. The number of rows in M up to n_o is equal to the number of events that can change the selected performance metric. ◆

The event counter $e(n)$ is an appreciation of the fact that rows are to be added in M dynamically with program execution time n, and the size of the performance data domain

at a time n_o is equal to $3e(n_o)$.

The size of the performance domain represented by matrix M is a dynamically growing quantity with time. It should be noted that the metric-based performance matrix M is already a simplification of the original trace data matrix T, defined in section 3.2 which has more dimensions (in general) compared to the matrix M. This fact underlies the unsuitability of the performance data domain for the purpose of analysis. Since each event occurrence can result in a three-dimensional row vector having all new entries, any type of graphical representation of the program execution will be difficult to comprehend for human senses, let alone to analyze it. The problem of many data dimensions is not unique to program performance analysis. Scientific visualization is also faced with the same problem, and methods such as "focusing and linking" have been proposed to reduce the number of dimensions of data to facilitate its visualization and analysis [14, 125]. Managing many dimensions also results in lack of scalability, extensibility, and efficacy of any analysis methods developed in the performance data domain. This provides the primary motivation of transforming the performance data domain to the analysis domains, which is discussed in subsequent sections.

4.2 Spatial Analysis Domain

One of the distinguishing factors between serial and parallel (or distributed) processing, both at hardware and software levels of system abstractions, is the spatial dimension of the computation. At the hardware level, the spatial dimension is introduced due to the involvement of multiple processors (and their interconnection network) to concurrently work on the same problem. At the software level, the spatial dimension comes into play by allowing multiple processes (or other software entities) to be executed on processors that are physically separate. These processes communicate via the interconnection network in order to accomplish specific programming objectives. There also may be multiple processes at a single physical processor.

The spatial dimension can reflect the physical or logical configuration of processors (or

processes) in the system. Interactions among processors during program execution give rise to changes in the states of the processors. Processor state could be a value based on the instantaneous status of a processor or it could be reflected by cumulative value of a metric at a specific instant of time. In either case, processor state, if obtained from matrix M, is a function of two discrete variables, instant of time (n) belonging to the given time-scale and spatial location (p) determined by the logical number assigned to the processor at the start of the program. To reduce the complexity of analyzing system states, we can focus on spatial dimension and hide the temporal dimension. That is, we can observe the system state in the "spatial domain" at a particular instant of time. While the metric-based performance data matrix M defines a three-dimensional subspace, the spatial domain is characterized by a two-dimensional subspace depicting the processor states at all the spatial locations involved. All the entries of M come from a real field.

Definition (Transformation to Spatial Domain)

Let T_1: $R^{e(n_o)\times 3} \rightarrow R^{P\times 2}$ transform $M \in R^{e(n_o)\times 3}$ to the spatial domain at a specific observation time n_0, such that:

$$T_1(M(n,p))\,|_{n=n_o} = S(p)\,|_{n=n_o} = \left[p_k\, m_k(n_o)\right] \tag{4.1}$$

for $k = 0,1,\ldots,P\text{-}1$, where P is the number of processors allocated for the program and $e(n_o)$ is the value of event counter at time n_o. ◆

Matrix S of dimensions $P \times 2$ represents the spatial domain. P is typically much smaller than $e(n_o)$, so the size of the data set is smaller. Also, P is a constant over time, so the size of the dataset in the spatial domain also remains constant over time. Matrix S represents the (logical) processor numbers in the first column and the states of the corresponding processors in the second column, at the observation instant n_0. States of all the processors at time n_o can be obtained from the state vector $v(n)$ that is updated for all $n \leq n_o$, in order to make this transformation work in practice. Matrix S does not take into account any physical or logical interactions that might be present among the processors during the program execution. In order to account for the spatial structure (physical or logical) of the system, we have to transform the matrix S into a digital image called a *performance*

image. This transformation is performed by a linear operation commonly known in the visualization discipline as an *image rendering* operation.

Definition (Performance Image)

A performance image is a representation of the state of all the processors in the system in a two-dimensional plane, at an observation time n_0. It is a digital image obtained by a *rendering transformation* applied to S to form a bound matrix G of dimensions $r \times c$, where total number of processors $P = r \times c$. The rendering transform is defined as:

$$S(p)\,|_{n_o} \rightarrow G(n_o) \tag{4.2}$$

where elements of G are color-table indices such that:

$$G(n_o) = \{g(i,j) \mid (g(i,j) \in R), g(i,j) = \alpha m_k(n_o)\} \tag{4.3}$$

where $i = 1,2,...,r$; $j = 1,2,...,c$; $k = 0,1,2,...,P\text{-}1$ and α is the ratio of the ranges of the available color index and the performance metric used for representing system state, specified by:

$$\alpha = \frac{g_{max} - g_{min}}{m_{max} - m_{min}} \tag{4.4}$$

Determination of the spatial location (i,j) in the performance image corresponding to the metric value of a particular processor k at the observation time n_o, given by $m_k(n_o)$, is dependent on the type of image rendering used (e.g., natural or gray-coded ordering of processors). The parameters r and c can be defined as:

$r = 2^{(d+1)/2}$ and $c = 2^{(d-1)/2}$ under the constraint that $P = 2^d$. ◆

In the case studies presented in Chapter 6, we have used natural ordering of the processors, as it depicted an aspect of the logical structure of the programs (data distributions). Various rendering paradigms can be selected by the user to suit the needs of a particular problem. The rendering operation is perhaps the most important link between the spatial image characterization and the logical structure of the program.

Theorem 4.1

A color in the performance image can be used to uniquely represent the numerical value of a performance metric, if and only if the color table has an infinite length.

Proof (if part):

According to the definition of a performance image $G(n_o)$, it is obtained by a linear mapping from the spatial domain matrix $S(p)$ evaluated at n_o. The spatial domain matrix $S(p)$ at time n_o consists of P rows, where P is the number of processors in the system. There are two columns of the matrix. The first column lists the logical number p of the processor, where $p \in \{0,1,...,P\text{-}1\}$. The second column consists of the value of an arbitrarily selected metric m at the processor whose logical number appears in the first column of the same row, at time n_o, where $n_o \in [0,n_{N\text{-}1}]$. Since the color table is indexed depending on the dynamic range of the performance metric (and the dynamic range of the color table itself), a performance metric value will be mapped to a particular color, as indicated by the definition of performance image. If the color table has an infinite number of entries, it can represent each value of the metric (a real number) by a unique color.

Proof (only if part):

Suppose that the color table does not have an infinite number of entries, i.e., $(g_{max} - g_{min}) < \infty$ where each color table index $g \in Z$. Suppose that g_1 and g_2 are two consecutive color table indices. Let m_1 and m_2 be two metric values which when mapped to a performance image according to the transformation specified by equation (4.2) get mapped to a continuous interval within $[g_1,g_2]$. Then, depending on the rounding technique being used (*truncation*, *rounding up*, *down*, etc.), both m_1 and m_2 will get mapped to either g_1 or g_2. Hence, the corresponding color in the performance image will not represent the value of the metric uniquely, if the length of the color table is not infinite. QED. ■

Corollary

If the color table does not have an infinite number of entries, then a range r of the metric values will be mapped to one particular color. In the limiting case, when $r \rightarrow 0$, it is possible to approximate the value of the metric by a color precisely (but not uniquely). ■

In a practical case, the color table can not have infinite length. However, if it is large enough compared to the dynamic range of a metric, it can represent the values of a metric to a fair degree of accuracy. The definition of performance image rendering seems to work well with the concurrent systems using *hypercube* and *mesh* interconnection topologies. Other types of renderings can be used in case of other topologies or to emphasize specific aspects of the logical interactions among the processors. Performance image was defined for a generic performance metric m which is a function of time and space variables. To be specific, if we suppose that m represents cumulative busy time of a processor at a given

observation instant n_o, then it can be defined as:

$$m_k(n_o) = \frac{1}{n_o}\{n_o - \sum_{k \in [0,n_o]} \Delta n_k(p)\} \tag{4.5}$$

for $p \in \{0,1,...,P\text{-}1\}$ and $\Delta n_k(p)$ is the time taken by the k-th message-passing event at the p-th processor. There are other metrics that can be used for generating the performance images, such as instantaneous processor states (at an observation time), cumulative communication volume and count, etc.

Transformation to the spatial domain converts the system state "snapshot" at a certain time to the performance image (matrix). Several image processing and analysis techniques can be used to glean more (high-level) information regarding the system state and program behavior as well as their variations over time. These techniques will be discussed in the subsequent chapter when we present the analysis techniques within specific domains.

4.3 Temporal Analysis Domain

Time is another important dimension of parallel program characterization, in addition to space. This dimension manifests itself differently at different levels of abstraction. For a programmer, the wall-clock execution time of a program is a means to characterize its performance. For a system analyst, the response time experienced by users during the execution of their jobs might help evaluate the efficiency of system software. Within a parallel program, the time taken by a processor to send or receive a message is associated with performance. Clock pulses from the local clock of a processor provide some notion of time. Therefore, regardless of the level of abstraction, performance of a parallel system can not be characterized without developing a time-scale to temporally distinguish among various events during the execution of a parallel program.

The temporal dimension reflects the random nature of occurrences of events during the execution of a parallel program. Although all the processors in a distributed system may have independent clocks, for our purposes, we leave the issues related with synchronization and determination of global time-stamps to the instrumentation system.

This is a classical problem in distributed processing, and instrumentation systems resolve this issue according to [51] to keep the "causality" intact among the message-passing events. Therefore, observations of a performance metric at equally-spaced intervals of time can be obtained from the trace data matrix M by transforming it from the performance data domain to the temporal domain.

The transformation to the temporal domain is accomplished by eliminating the spatial dimension from matrix M. The elimination of the spatial dimension results in a matrix with two columns: the first column represents time indices of events of interest which are not equally spaced, and the second column represents a time-series. We refer to this transformation as an *event domain* transformation, as the resulting matrix with two columns indicates the time of occurrences of events that cause changes in the value of the performance metric. The event domain will be discussed in more detail in the next section, and matrix M is transformed to the event domain here as an intermediate step in the transformation to the temporal domain. This transformation is accomplished with the help of state vector v(n), as in the case of spatial domain transformation.

Definition (Transformation to Event Domain)

Let T_2: $R^{e(n)\times 3} \rightarrow R^{e(n)\times 2}$ be the event domain transformation, such that:

$$T_2(M(n,p)) = M_e(n) = \begin{bmatrix} n_0 & x(n_0) \\ n_1 & x(n_1) \\ \dots & \dots \\ \dots & \dots \\ n_{N-1} & x(n_{N-1}) \end{bmatrix} \tag{4.6}$$

where $x(n)$ represents the values of the performance metric $m(n,p)$ which are obtained from the state vector $v(n)$ after the elimination of the spatial dimension by an appropriate reduction technique. ◆

The elimination of the spatial dimension using reduction is mainly dependent on the nature of the performance metric being traced by the state vector $v(n)$ as matrix M is processed. For instance, if $x(n_i)$ represents the fraction of the system being utilized (busy) at an instant of time n_i, then it can be specified by:

$$x(n_i) = \frac{1}{P}\begin{bmatrix}1 & 1 & \dots & 1\end{bmatrix}_{1\times p} v(n_i)$$

$$= \frac{1}{P}\begin{bmatrix}1 & 1 & \dots & 1\end{bmatrix}_{1\times p} \begin{bmatrix} v_0 \\ v_1 \\ \dots \\ \dots \\ v_{p-1}\end{bmatrix}$$

$$= \frac{1}{P}\sum_{k=0}^{p-1} v_k(n_i) \tag{4.7}$$

for $i = 0,1,...,N\text{-}1$. This equation holds when the metric $m(n,p)$ is used to represent the status of the processor p at time n by the state vector $v(n)$, where elements $v_k = 1$ mean that the processor is busy and $v_k = 0$ mean the processor is idle for $k = 0,1,...,P\text{-}1$.

The sequence $\{x(n)\}$ constitutes a time-series whose indices n are event numbers (and not time). These event numbers correspond to the time-stamps given by the first column of matrix $M_e(n)$. These time-stamps are not equally spaced as they correspond to the times of occurrence of specific events. The size of the dataset has been reduced from a $e(n) \times 3$ dimensional matrix in the performance data domain to a $e(n) \times 2$ dimensional matrix in the event domain, or to $e(n)$ dimensions if the event domain time-series $x(n)$ is considered. Although, the size of $x(n)$ is increasing over time (i.e., it is time-dependent), the transformation process has resulted in a mathematical form that is more convenient and powerful for analysis, i.e., a time-series. We will go one step further to model this time-series as a discrete-time stochastic signal, as it conforms to the definition of a discrete-time stochastic signal: *a physical quantity that varies with time, space, or any other independent variable whose past, present and future values can not be determined precisely by using a mathematical expression* [84]. This signal will be termed as an *event domain performance signal.* This model enhances our ability to process the time-series observations by using several well known digital signal processing (DSP) techniques that will be discussed in the next chapter.

For the purpose of temporal domain analysis, the time-scale n has to be equally-spaced.

Therefore, we transform the matrix $M_e(n)$ from the event domain to the temporal domain and the resulting time-series will be denoted by $w(n)$ to distinguish it from the event domain representation $x(n)$.

Definition (Transformation to Temporal Domain)

Let T_3: $R^{e(n)\times 2} \rightarrow R^{n_N}$ be the temporal domain transformation, such that:

$$T_3(M_e(n)) = w(n) = \begin{bmatrix} A(n_0) \\ A(n_1) \\ \dots \\ \dots \\ A(n_{N-1}) \end{bmatrix}_{n_N \times 1} \tag{4.8}$$

where $A(n_i)$ are block matrices given by:

$$A(n_i) = \begin{bmatrix} x(n_i) \\ x(n_i) \\ \dots \\ \dots \\ x(n_i) \end{bmatrix}_{(n_{i+1}-n_i)\times 1} \tag{4.9}$$

for $i = 0,1,...,N\text{-}2$ and $A(n_{N-1}) = x(n_{N-1})$. ◆

The sequence $\{w(n)\}$ is a time-series and again will be treated as a discrete-time stochastic signal, called a *temporal domain performance signal*, so that we could analyze it using DSP techniques.

Definition (Performance Signal)

An event domain or time domain performance signal is a discrete-time stochastic signal, representing the values of a certain performance metric at discrete instants of time during the execution of an instrumented program. A performance signal will be denoted by $w(n)$ when it is a time-series obtained by temporal domain transformation and it will be denoted by $x(n)$ when it is a time-series obtained by the event domain transformation. ◆

The transformation to the time domain via the event domain has certain practical advantages. As described in the previous chapter, the instrumentation technique relevant to this work is *event-driven tracing*, i.e., the data logged to the trace file are dependent on

the occurrence of events of interest. This strategy reduces the amount of data to be logged and stored during the program execution (in some cases) compared with the *sampling* technique where data are logged at equally spaced intervals, regardless of the occurrence of events of interest. The time of event occurrence and the metric value are later used (as defined above) to transform the data to the temporal domain only when analysis is to be carried out.

Theorem 4.2

A performance signal can be used to represent the dynamic behavior of a program being executed on a parallel computer system.

Proof:

The proof follows directly from the definitions of dynamic behavior and performance signal. The transformation to the performance signal through the event domain involves elimination of the spatial dimension by using an appropriate reduction technique on the time-series data which is spatially distributed. This gives rise to another time-series that represents the observations regarding the state of the whole system. By definition, this time-series represents the dynamic behavior of the program on the given parallel system. QED. ■

Various signal processing techniques are applied to the time-series $w(n)$ for its analysis. These techniques are discussed in the next chapter.

4.4 Event Analysis Domain

Performance data are generated in response to the occurrence of specific types of events during the execution of an instrumented program. The occurrence of events and consequent changes of system states can be analyzed more appropriately in what has been referred to as the event domain [1, 68]. The aspects of the program and system behavior that are related with the global state transitions are best explained when characterized without any dependence on the spatial dimension and with minimal dependence on the temporal dimension. The system states may include at least the busy and idle states of the system, and the event domain can characterize the transitions between these states.

The transformation to the event domain, denoted by T_2, was defined in the previous

section. The spatial domain has to be eliminated completely so that we can look at the system from a global perspective. The dependence on the temporal dimension is minimal. The first column of $M_e(n)$ lists the time indices (time-stamps) corresponding to the occurrences of events that change the system state. The second column denoted by $x(n)$ is a discrete-time stochastic signal that characterizes the event domain. The only difference between the event domain representation $x(n)$ and the temporal domain representation $w(n)$ is that the event domain represents the state transitions and not the transition (or holding) time between the states as shown by the temporal domain representation. We analytically formulate the difference between the two types of performance signals with the following theorem.

Theorem 4.3

An event domain performance signal is equivalent to a temporal domain performance signal except for the state holding times that are eliminated in the event domain performance signal.

Proof:

Consider an event domain performance signal $x(n)$ obtained from the matrix $M_e(n)$ defined as an intermediate step for the transformation to the temporal domain. Suppose that $x(n)$ consists of N samples $\{0,1,2,...,N\text{-}1\}$ that correspond to N time-stamps $\{n_0, n_1, n_2,..., n_{N-1}\}$, respectively, where events of interest occurred. Event and temporal domain performance signals can be obtained by working through the transformation from $M_e(n)$ to $x(n)$ and then to $w(n)$.

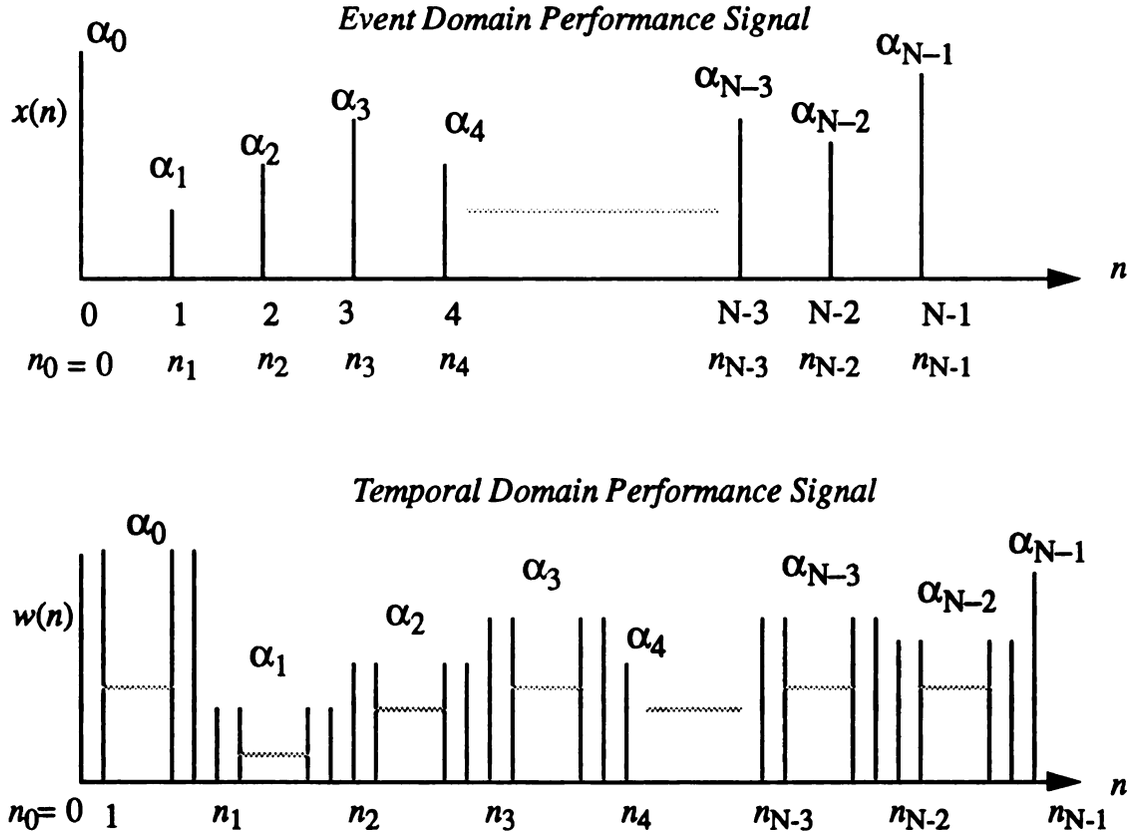


Figure 4-1. Differences between Event and Temporal Domain Performance Signals.

Let $\delta(n)$ be the discrete-time unit impulse function and $u(n)$ be discrete-time unit step function. Then the temporal domain performance signal can be represented as:

$$w(n) = \alpha_0\{\delta(n)+u(n-1)-u(n-n_1+1)\}+\alpha_1\{\delta(n-n_1)+u(n-n_1-1)-u(n-n_2+1)\}+$$
$$\alpha_2\{\delta(n-n_2)+u(n-n_2-1)-u(n-n_3+1)\}+\alpha_3\{\delta(n-n_3)+u(n-n_3-1)-u(n-n_4+1)\}+\text{---}+$$
$$\alpha_{N-2}\{\delta(n-n_{N-2})+u(n-n_{N-2}-1)-u(n-n_{N-1}+1)\}+\alpha_{N-1}\{\delta(n-n_{N-1})\}$$

$$w(n) = x(n)+\alpha_0\{u(n-1)-u(n-n_1+1)\}+\alpha_1\{u(n-n_1-1)-u(n-n_2+1)\}+$$
$$\alpha_2\{u(n-n_2-1)-u(n-n_3+1)\}+\alpha_3\{u(n-n_3-1)-u(n-n_4+1)\}+\text{-------}+$$
$$\alpha_{N-2}\{u(n-n_{N-2}-1)-u(n-n_{N-1}+1)\}$$

$$w(n) = x(n)+\sum_{i=0}^{N-2}\alpha_i\{u(n-n_i-1)-u(n-n_{i+1}+1)\}$$

where $n_0 = 0$ as shown in Figure 4-1. Therefore, we can express the temporal domain performance signal as:

$$w(n) = x(n) + \prod(n) \tag{4.10}$$

where $\Pi(n)$ is a collection of N-1 "windows" of state holding-times, each of length $n_{i+1} - n_i - 1$ for $i = 0, 1, 2,..., N$-2. This is exactly what is missing from the event domain signal. These windows contribute to the expansion of the time-scale in the case of temporal domain performance signal, whereas the event domain performance signal consists of the "jumps" of this temporal signal. QED. ■

This theorem contributes to our understanding of the difference between temporal and event domain performance signals. This is useful information because the event domain performance signal corresponding to a given temporal domain performance signal consists of a set of samples that is orders of magnitude smaller. However, the dynamic behavior is still shown by the event domain signal even though the state holding-times are missing.

Theorem 4.4

Temporal domain information can be recovered from the event domain matrix $M_e(n)$.

Proof:

The proof follows from the definition of the event domain transformation of the metric-based performance matrix $M(n,p)$ to $M_e(n)$. Let $x(n_o)$ and $x(n_o+1)$ be two consecutive samples of an event domain performance signal. In order to determine the relative distance between the two samples on a "true" time-scale, we use n_o and n_o+1 as indices to the first column of the matrix M_e(n) and determine n_{n_o} and n_{n_o+1} which are the corresponding time-stamps. The size of the holding time window between these two samples is equal to $n_{n_o+1} - n_{n_o} - 1$ with an amplitude equal to $x(n_o)$. Therefore, all the temporal domain information has been recovered. Similarly, for all other event domain performance signal samples, we can use the same conversion procedure to retrieve the temporal information. Hence, the temporal domain performance signal can be recovered from the event domain matrix $M_e(n)$. QED. ■

4.5 Frequency Analysis Domain

When the temporal and event domains are used to analyze the dynamic behavior of the program with the help of a number of signal processing techniques, it is only natural to encounter the frequency domain. Since we have clearly shown the difference between the temporal and event domains, we will mention only the temporal domain in this section (to be consistent with the peculiar notion of time and frequency domains found in signal

processing literature). The discussion equally applies to the event domain signals.

Time and frequency domains complement each other to analyze discrete-time signals. As we have characterized the temporal (and event) domain using discrete-time performance signals, we need to design various systems (filters) to process and analyze it. These systems are easier to design in the frequency domain under various circumstances. Therefore, the frequency domain is useful for multiple-domain analysis methods that will be discussed in the next chapter.

The idea of specifying the temporal behavior of a program in the frequency domain has already been used [1]. Besides the advantages in filter design, a frequency domain representation provides a "graphical signature" of the types and extent of variations present in the performance signal, known as the *spectrum* of the signal. A user might not be able to extract this information merely by looking at a performance signal.

The notion of "frequency" of the performance signal needs to be clarified as it is not a signal generated by a "physical" phenomenon that might consist of a range of frequencies and is sampled at an appropriate frequency to convert it to a discrete-time signal. A performance signal is an inherently discrete-time signal, and the resolution of the event logs (time-index) is determined by the resolution of the instrumentation system. This resolution might be equal to the time-period of a local clock or it might be several multiples of that time-period. In general, if $T_{\min}$ is the minimum possible time between two successive (instrumented) events (in seconds) that can be registered by the instrumentation system, then the maximum frequency (in radians) of the resulting performance signal is given by:

$$\omega_{max} = \frac{2\pi}{T_{min}} \tag{4.11}$$

Definition (Transformation to Frequency Domain)

Let T_4: $R^n_N \rightarrow C^n_N$ be a linear operator, such that:

$$T_4(w(n)) = W(K) = \sum_{n=0}^{n_N-1} w(n)\, e^{-j\frac{2\pi}{N_N}nK}, \qquad \text{for } K=0, 1, \ldots, n_N-1 \tag{4.12}$$

The linear operator that converts the real time-series (performance signal) to a complex subspace of the same dimensions, is commonly known as the *discrete Fourier transform* (DFT). ◆

A fast Fourier transform (FFT) algorithm can be used to efficiently transform the performance signal to the frequency domain. The determination of the spectrum leads to various digital filtering techniques to selectively reject a band of frequencies to aid a particular type of analysis [71, 84].

The characterization of the four analysis domains and the transformation to those domains comprise the starting point of multiple-domain performance analysis. We had mentioned the application domain in Chapter 2 which is of interest but difficult to incorporate in trace-based analysis. We briefly discuss the nature of this domain in the following section.

4.6 Visualization in the Application Domain

We have formalized the analysis domains that are of interest in the context of multiple-domain analysis approach. However, it will be interesting to discuss another domain that is useful for presenting the dynamic behavior of a program on a particular system.

The application domain is an abstraction rather than a finite dimensional vector space, per se. This domain is associated with the development of an application program and uses that information to depict the behavior of the program [109]. It does not rely on the trace data. Instead, the programmer must develop a display that matches his/her mental view of the program. Then information about the dynamic behavior of the program is passed to the display directly from within the program, i.e., the application domain. Therefore, in order to use the application domain, the activities of program development and visualization display development are combined. There are at least two advantages that this technique can yield:

1. Visualization displays provide the application domain information to the user, and the user need not go through the additional process of linking the information from a generic display to the behavior of the application program.
2. This technique is more suitable for debugging parallel and distributed programs rather than evaluating their performance.

Visualization in the application domain is closer to being an art as the programmer relies on his/her imagination to create effective displays for visualization. The user does not need to rely on the observed performance data to glean information about the higher-level programming abstractions. For instance, if a sorting algorithm is visualized, the application domain visualization should depict the program as it accesses different data values, according to the programmer's mental model about the computation [13, 109]. On the other hand, the trace-based methods that are the focus of this work require the user to link the observed performance that is displayed with the behavior of a program.

A reason for considering the application domain here is to appreciate the fact that the information from the application domain is always useful, regardless of the performance analysis or program debugging methodology to be used. If a programmer has *a priori* knowledge about the behavior of a program, then analysis methods can serve the purpose of either confirming or contradicting that knowledge. In either case, this information will be helpful to tune the performance of the program. For example, we have applied *a priori* knowledge of the application domain in case studies to verify the estimates of loop iteration times, which will be presented in the next chapter. However, it is not trivial, if at all useful, to define and characterize the application domain rigorously as for the other analysis domains. Therefore, this subject is not pursued any further in this thesis.

The definitions and characterizations of multiple analysis domains in this chapter provide a basis for the analysis presented in the next chapter. These domains allow the analysis of various aspects of program behavior and system performance where it can be done most effectively. For instance, system states can be analyzed effectively in the spatial domain, and dynamic behavior can be analyzed in the temporal or event domain. Information about program behavior from one domain typically supplements the information from another

domain. The analysis in multiple domains eventually contributes to the programmer's understanding of the global picture of a program's performance.

Chapter 5

Multiple-Domain Analysis Methods

The transformations to various analysis domains presented in the previous chapter is the basis of the multiple-domain analysis approaches presented in this chapter. We will focus on the presentation of specific analysis techniques that are applicable to the performance data in each of the analysis domains and on the information that a domain can yield about the program behavior. It is important to point out that a particular analysis method is useful in the context of program performance analysis if it can extract any useful information about the higher-level abstractions of the program behavior on a parallel computer system. The analysis techniques that will be presented in this chapter are commonly used in digital signal and image processing and statistical (time-series) analysis disciplines. Once performance data are transformed into a particular form, in a particular domain, it is often tempting to apply available techniques exhaustively. Although the analysis might be valid theoretically, its contribution towards a better understanding of the program behavior might be minimal. Therefore, we will restrict ourselves to the methods that are appropriate in this context. In fact, identifying a set of appropriate methods and the particular situations in which to apply them has only been partially done in this thesis. Nevertheless, this approach helps meet the objective of making the performance analysis of medium- and large-scale parallel systems more rigorous, scalable and extensible.

Analysis methods in each of the spatial, temporal and event domains will be presented separately, starting from the domain-specific representations developed in Chapter 3. As the frequency domain analysis mainly complements the temporal and event domain analyses, we limit our discussion of the frequency domain. After separately presenting the analysis methods and their usefulness in multiple domains, we will discuss their contributions towards the analysis of the behavior of a program. We then prepare a strategy to analyze a given program by multiple-domain analysis methods.

5.1 Analysis in the Spatial Domain

The spatial domain was characterized in Chapter 4 by performance images at particular observation times during the program execution. The analysis methods used in this domain have been derived from this characterization using typical image processing techniques. The objectives of analysis in the spatial domain can be listed as follows:

- support of visual analysis of states of the system by computational methods;
- exploitation of the graphical representation of the system states by associating the analysis with display;
- representation of data distribution in the context of a programming paradigm, such as SPMD (single-program, multiple-data);
- representation of data movements during the program; and
- detection of changes in the system states.

These objectives will be accomplished by applying appropriate image processing techniques on the performance image matrix that was denoted by $G(n_0)$ at an observation time n_0 during the execution of the program. These techniques include analysis of *image statistics*, *image subtraction*, *thresholding* and various *image enhancement* operations. These methods and their application to the analysis of the spatial behavior of a program is explained in this section.

5.1.1 Representation of Macroscopic States

The macroscopic state of the system at a particular time during the execution of a program contributes towards the understanding of the overall behavior of the program. The following theorem states that the spatial domain characterization using performance images is a valid representation of the macroscopic system state at a particular instant of time.

Theorem 5.1

Performance image at a certain instant of time represented by the matrix $G(n_0)$ describes the state of the system in a two dimensional grid where individual processor states are mapped according to the rendering operation used for this purpose.

Proof:

Let $G(n_o)$ be the performance image matrix at time n_o which can be used to represent corresponding performance image (proved by the following argument). As mentioned in Chapter 4, if the number of processors $P = rc$, then the image matrix is given by:

$$G(n_o) = \begin{bmatrix} g_{11} & g_{12} & \cdots & g_{1c} \\ g_{21} & g_{22} & \cdots & g_{2c} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ g_{r1} & g_{r2} & \cdots & g_{rc} \end{bmatrix} \tag{5.1}$$

[31] lists three conditions that must be fulfilled by a matrix to be a *bound matrix*, which can be transformed to a digital image. These conditions are:

1. $g_{ij} \in$ R (set of real numbers)
2. $1 \leq i \leq r$, $1 \leq j \leq c$ and
3. $r, c \in$ Z (set of integers)

Since $G(n_o)$ fulfills these conditions, it is a *bound matrix* and it can be represented as a performance image. Each of its elements g_{ij} concerns the following:

- g is an index to a color table corresponding to the value of a performance metric that has been transformed to the spatial domain by the spatial domain transformation (defined in Chapter 4), and
- the pair (i,j) specifies a spatial location of a processor in a two-dimensional performance image, as defined by the rendering transform (in Chapter 4).

When this matrix is transformed to a digital image, then it follows from the definition of the system state that it represents the (macroscopic) state of the system at a time n_o. QED. ■

A macroscopic display is the first step in a top-down hierarchical approach to performance analysis. This is a particularly useful approach for a large-scale parallel system having several thousands of processors. There are similar "matrix displays" in conventional analysis tools but they are not adequately scalable. Performance images work in this case due to the image matrix structure defined in Chapter 4. Depending on the structure of a program, the representation of macroscopic states provides the following information to the user:

- Performance images can represent the *data distribution* of a data parallel program. This type of programming paradigm is especially suitable for computationally intensive numerical and signal processing problems. Such computations inherently use matrices to represent the data. The program distributes the data (usually elements or blocks of the data matrix) to various processors. All processors execute the same instructions on their local data values. If the manner in which the program-data were distributed among the processors is captured during the rendering operation, then the pixels of the performance image represent this logical information about the data distribution.
- A series of performance images taken over the execution time of a program can represent the *spatial movement* of the computation. Even in the case of a well-structured program, there might be several phases of the program that are data-dependent and have to be executed only by specific processors. In various computation-intensive problems, the direction of movement of the computation could be specified. A series of performance images, for example, can show the movement of utilization "hot spots".
- When a parallel program has little, if any, "regular" structure, performance images might be useful to understand the behavior. However, in this case, there is no "coupling property" between a performance image and the computation (such as the data distribution or movements). The physical structure depicted by the performance images will have to be linked to the program behavior "manually" by the user. In this case, patterns in one or a series of performance images should not be thought as representative of logical structure of the program, because this might prove misleading.

5.1.2 Change Detection via Image Subtraction

Image subtraction is a simple yet powerful technique (in a carefully controlled imaging situation) to compare two complicated images. Two images are aligned and the arithmetic operation of subtraction is performed on their bound matrices. The difference image is then enhanced to detect changes in a dynamic situation or to compare an image with a template. Image subtraction is used for imaging of blood vessels using X-rays; detection of missing components from a circuit board; automated inspection of printed circuit boards; security monitoring based on motion detection; monitoring growth patterns of urban areas; target detection from radar images; weather prediction from satellite pictures [57], and so on.

We have applied an image subtraction technique on performance images for two purposes:

1. Detection of changes in system states over time, and
2. Matching spatial patterns of two performance images.

These techniques are explained in the following sections.

5.1.2.1 Change Detection

Changes in system states between two discrete instants of execution time of a program can be detected by subtracting the performance images at those instants and enhancing the difference image. If $G(n_a)$ is a performance image matrix representing the macroscopic state of the system at time n_a and $G(n_b)$ is another image matrix (of the same dimensions) representing the system state at time n_b, such that $n_b > n_a$, then their difference image is given by:

$$\Delta(n_a,n_b) = G(n_b) - G(n_a) \tag{5.2}$$

where each of the elements in the difference image matrix $\Delta(n_a,n_b)$ is the difference of the corresponding elements of $G(n_a)$ and $G(n_b)$. The range of values of the elements of the difference image matrix δ_{ij} is given by: $-g_{max} \leq \delta_{ij} \leq g_{max}$, where g_{max} is the maximum value of the available color table index. Usually, the range of the elements is "stretched" by mapping it to the full range of the available color table index, as the two images being subtracted might be quite similar. This enhancement operation facilitates the visual detection of the changes between the two performance images.

Detection of changes between dynamic macroscopic states of the system at two different time instants has been referred to as *analysis by small multiples* in the literature on graphical representation of information [116, 117]. A series of performance images can be taken over the execution time of a program, and two images can be subtracted to show the spatial difference patterns of a metric from one time to the next. When the program has a regular structure, the difference of spatial patterns shows the "flow" of computation among the processors. On the other hand, if the problem is not well-structured, a user has to exert more effort to correlate the dynamics of the spatial patterns with the program behavior.

5.1.2.2 Spatial Pattern Matching

Image subtraction can be used to match a particular image with another image having known patterns in it (i.e., a template) to compare the two. This technique is useful when

the performance image has intricate patterns that are hard to detect visually. A template of known "correct" spatial patterns can be subtracted from a performance image to determine the degree of mismatch between the two. If the digital (performance) image of dimensions $r \times c$ is represented by $G_1(i,j)$ and a template performance image is represented by $G_2(i,j)$, then the mismatch energy provides a quantitative measure of differences between the two, and is defined by [57] as:

$$\sigma^2 = \sum_{i=1}^{r} \sum_{j=1}^{c} [G_1(i,j) - G_2(i,j)]^2 \tag{5.3}$$

The mismatch energy is minimum when the performance image and the template match closely with each other. This technique is useful for both performance and program debugging. If the correct behavior of a program on a system is already known and can be characterized as a performance image at a logically appropriate instant of time, then it can be used as a template. The performance image from another version of the same program at the same instant of time should closely match with the template. If there is any marked difference, then it will be helpful to spatially locate the anomaly that could be further traced in the program. This process is particularly useful for MPP systems where visual or textual analysis of any such anomaly is not very effective.

5.1.3 Thresholding

A thresholding transformation is applied to digital images to identify various features in the image. Since a performance image shows the spatial distributions of a performance metric, the thresholding technique can segment this image to emphasize the areas of high and low computation or communication activity (depending on the metric). The thresholding transformation is a point-operation which is applied to each element of the performance image matrix $G(n_o)$ to transform it to $G^t(n_o)$. If Th represents a threshold function, then the thresholding transform is specified by:

$$Th: G(n_o) \rightarrow G^t(n_o) \text{ such that } g_{ij} \in G(n_o) \text{ and } g^t{}_{ij} \in G^t(n_o). \tag{5.4}$$

There are various ways to define thresholding. We will define the thresholding operator in two closely related ways, but both will provide different results.

5.1.3.1 Identification of Hot-Spots of Activity

By selecting a suitable threshold value, the higher values of the color index can be isolated from the rest of the performance image. This operation is defined in terms of the performance image matrix elements, before and after thresholding, as:

$$g_{ij}^{t} = \begin{cases} g_{ij} & \text{if } g_{ij} \geq th \\ g_{min} & \text{Otherwise} \end{cases} \tag{5.5}$$

where th is the value of the threshold. Threshold need not be a single constant. It can be defined as a range of values such that $th \in \{th_{min}, \ldots, th_{max}\}$ where th_{min} is the specified minimum value of the threshold and th_{max} is the maximum value. The resulting image after thresholding will have only those color indices that are equal to or greater than the threshold value specified. All other spatial locations will have minimum value of the color table index. This technique can be used to isolate those spatial locations where the value of the metric is beyond the specified threshold. Therefore, if the metric represents the computational load of each processor, then this technique can identify the "hot-spots" of computation that might be a cause of an imbalance in the work-load distribution among the processors. On the other hand, if the metric is related to the message-passing actions performed by each processor, then this technique can isolate the hot-spots of communication activity to or from a certain ensemble of processors. Such information might prove valuable for the programmer to optimize the program in such as way that the computation and/or communication load on all the processors is balanced.

5.1.3.2 Identification of Types of Change

The thresholding operation can be used in conjunction with image subtraction to not only locate the changes but also to find whether the values of the metric are increasing or decreasing with time at various spatial locations. Suppose that the image matrix g_{ij}

contains a difference image between two performance images taken at two different instants of time. Then the thresholding transform can be defined as:

$$g_{ij}^{t} = \begin{cases} g_{max} & \text{if } g_{ij} \geq th \\ g_{min} & \text{Otherwise} \end{cases} \tag{5.6}$$

We will refer to this operation as *binary thresholding*, as there are only two color index values in the resulting image. This operation narrows the focus of the analysis to only two types of changes over time, for example to positive changes (i.e., value of performance metric increases) or negative changes (value decreases) over time. If we take a series of performance images during the execution time, we can subtract the successive images and apply binary thresholding to each difference image. The resulting series of binary thresholded images will show the decreasing or increasing activity related to computation or communication at each processor. This technique is useful to find movement of data and/or work when the program has a regular structure that is manifested in the performance image.

5.1.4 Image Enhancement

Image enhancement involves operations that improve the display of an image without distorting the features of the image. When image processing operations are being used for the purpose of performance analysis, then it is natural to use the usual image enhancement capabilities of the image processing tool. Enhancement operations include sharpening of image features such as edges, boundaries, or contrast; noise reduction; filtering; interpolation; and magnification (commonly known as zooming). The inherent information content (determined by the original image matrix) does not increase by the image enhancement operation. These operations only enhance the efficacy of the visual analysis by a human user.

The use of image processing techniques and tools for the purpose of analyzing program performance in the spatial domain provides some additional information, by default, such as image statistics. Image statistics complement the analysis performed by the image

processing techniques described above. Figure 5-1 summarizes the overall strategy of using spatial domain analysis by using image processing techniques. Performance data is modeled by metric-based performance data matrix *M*, which is then transformed to the spatial domain. The spatial domain matrix undergoes a rendering transformation to an image matrix *G* which can be represented as an image. Performance analysis methods have been divided into two categories: image statistics and image analysis.

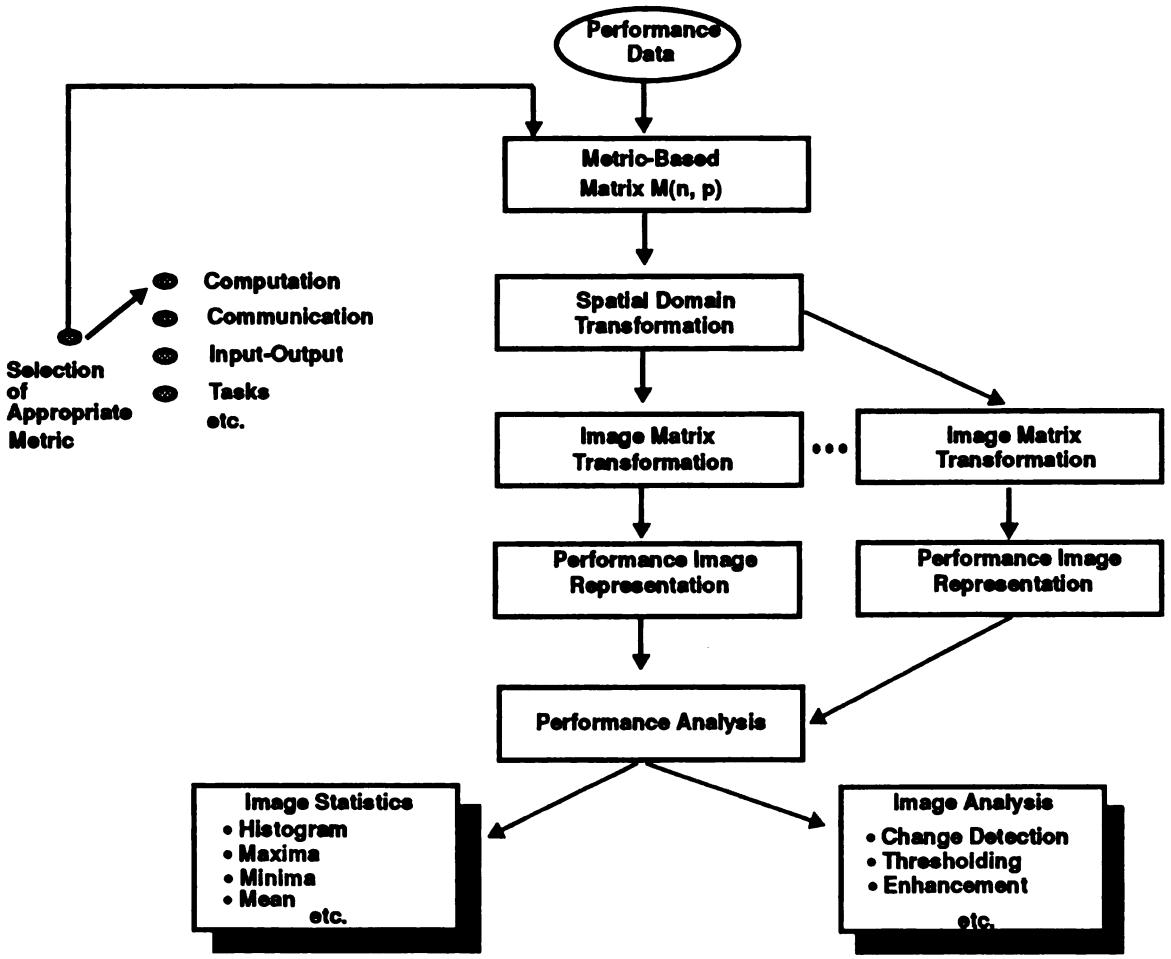


Figure 5-1. Application of image processing to performance analysis in the spatial domain.

5.2 Analysis in the Temporal Domain

Performance analysis in the temporal domain is based on the discrete-time stochastic signal (or time-series) representation of a performance metric. Various digital signal processing (DSP) techniques can be applied to analyze the pattern in the time-series to

obtain high-level behavioral information about a program. The temporal domain representation of the performance data as a time-series, denoted by $w(n)$ in Chapter 4, can be considered as a realization of a stochastic process. In general, this process is non-stationary [17] and some technique of imposing stationarity has to be adopted for further statistical analysis. This is a future direction of this research work. Even without specifying a suitable statistical model for the underlying stochastic process, important information about the temporal behavior of the high-level abstractions of a program can be obtained by applying suitable DSP techniques to the time-series.

5.2.1 Analysis of Trends and Phases

A *phase* of a program has different meanings for a programmer and for "automatic" analysis of the time-series. Usually a phase of a program is a collection of code that performs a specific logical task among a set of tasks required for the whole application. These phases occur sequentially during program execution. From an analysis point of view, a phase has been defined as a part of the time-series that exhibits stationarity [17, 73], although the overall time-series is not stationary. Therefore, such phases might not be immediately discerned due to the presence of large amount of variance. The objective of analysis is to remove this "noise" that hides the presence of these phases. The time-series is "smoothed" using methods such as moving averages and least-squares polynomial fitting by [17, 73]. The smoothing process results in the identification of *stationary phases*, which are periods of time when the time-series is smoothed equal to its mean during that period (i.e., variance is removed completely from a stationary segment of the time-series). This method works well for prediction of certain temporal properties of the program [17], but in general even the developers of this approach accept that the automatic identification of phases does not always work and manual detection of phases of a program might be more accurate in some cases [74].

Despite the limits of the automatic identification approach, the smoothing of the performance signal is still useful for analyzing the trends. The reduction of variance from the signal gives a "top-level" macroscopic view of the behavior of the program. This will

suppress the low-level details of the temporal changes of a metric, but on the other hand, it will provide the overall trends of that metric. Trends may be difficult to discern visually, especially in the case of complicated patterns resulting from a large amount of message-passing among processors. A user can always return to the original performance signal (or any other low-level view) to glean more "microscopic" information as needed.

For the purpose of analysis of trends of a performance metric, we have adopted smoothing operation by applying a digital moving averages filter to the performance signal. This technique is not different from what has been used by [17, 73]. The only difference is the application of DSP approach with the performance data transformed to a discrete-time signal. This approach is advantageous in the sense that the processing of large amounts of data is a practical problem in DSP and is solved either by resampling the data at a lesser sampling rate (decimation) or by operating on a block of data at a time (windowing). Such techniques reduce the data to be processed at a time but preserve the requisite information, therefore, are useful for analyzing long-running programs.

The digital moving averages filter can be specified by its difference equation as:

$$y(n) = \frac{1}{N} \sum_{k=0}^{N-1} w(n-k) \tag{5.7}$$

where $y(n)$ is the moving average at discrete-time n. Figure 5-2 shows a block diagram of the implementation of this filter. This is a finite impulse response (FIR) and causal filter,

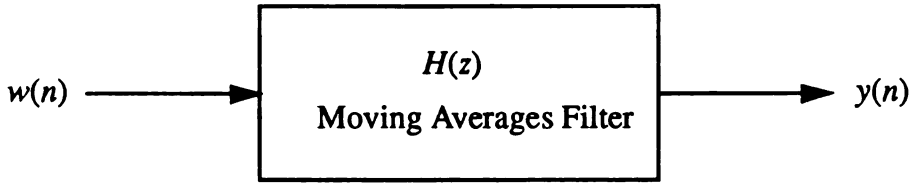


Figure 5-2. Smoothing of the performance signals by a digital moving averages filter.

with N-taps all having the same weight (1). There are different strategies to determine the weights, such as giving the most recent observation the maximum weight and progressively lesser weights for the past observations, which imposes a "forgetting factor"

on the series [5, 126, 127]. Since we do not claim to have adopted a particular model for the underlying stochastic process, we have empirically adopted uniform weights for all the lags. This filter can be specified by its system function $H(z)$ which is determined by evaluating the z-transform of the input-output difference equation and is given by:

$$H(z) = \frac{Y(z)}{W(z)} = \frac{1}{N}\sum_{i=0}^{N-1} z^{-i} = \frac{1}{N}\frac{1-z^{-N}}{1-z^{-1}} \tag{5.8}$$

where $W(z)$ and $Y(z)$ denote the z-transforms of $w(n)$ and $y(n)$, respectively. This filter is also known as a *comb filter*. The design parameters of the filter are obtained from this system function. The frequency response of this filter, shown in Figure 5-3, emphasizes

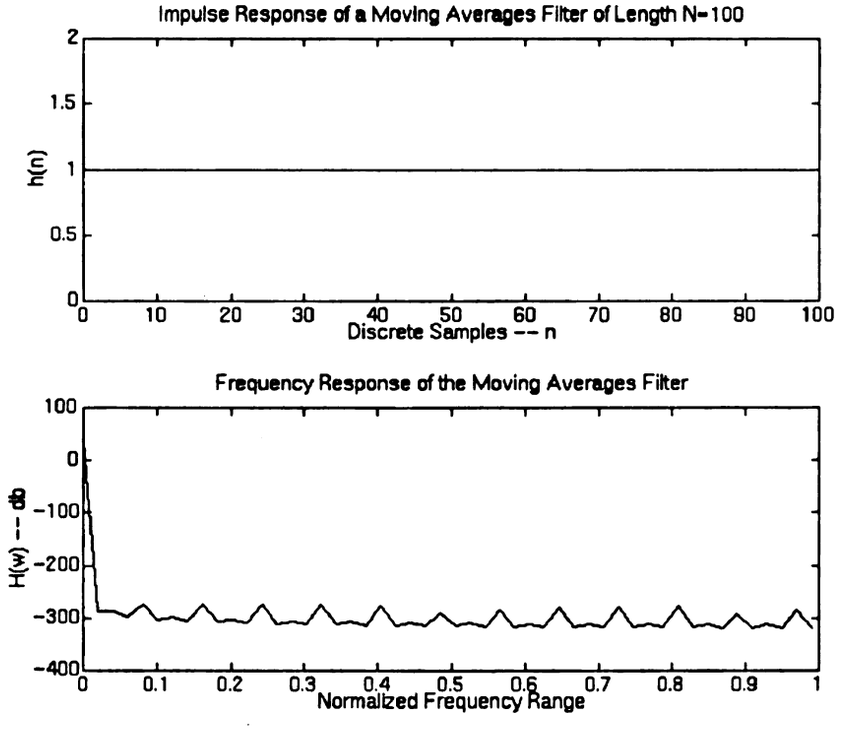


Figure 5-3. Impulse response of the moving averages filter.

the low-pass characteristics of this moving averages filter that consequently smooths the signal by reducing the variations present in it. This low-pass filtering operation eliminates the excessive fluctuations that tend to hide the trends in the data. The order of the moving averages filter is arbitrarily chosen as 100 in the case shown in Figure 5-3. The removal of

irregular patterns makes it possible for the user to visually appreciate the dominant patterns. The disadvantage of using a moving averages filter include the loss of data at the starting point and the effect of the extreme values on the average value. However, the filter adequately satisfies our objective of representing the trends in the performance signal.

5.2.2 Analysis of Repetitive Patterns

Repetitive patterns are expected in a performance signal due to the presence of iterative code structures such as loops in a program. These repetitive patterns can be observed as similar segments of the signal repeating over time, in response to each iteration of the loop. This periodic behavior can be observed using any conventional time-series plot available in a performance visualization tool (e.g., a space-time diagram). This periodicity, however, is not automatically obvious in all cases due to:

- Asynchronous execution in an SPMD programming paradigm, where some processors might not be working on the instructions within a loop.
- Excessive communication activity, which makes the plot too "dense" to identify the presence of any repetitive patterns.
- Large number of data samples in the display, which also makes it difficult to visually identify the presence of any repetitive patterns.

A widely used method to determine the repetitive (periodic) behavior of a stochastic signal (time-series) is by using an estimate of autocorrelation of that signal [84]. The exact (statistical) autocorrelation is not trivial to be determined for any stochastic system, therefore, temporal observations are used to estimate the autocorrelation. There are some underlying assumptions, such as stationarity and ergodicity which have to be assumed for such calculations [82]. It has been shown by [17] that the time-series data obtained from the dynamic behavior of a program (performance signal) consists of stationary phases (or can be made stationary using smoothing techniques). Since we are considering a very small segment of the whole signal that represents a part of a "phase" of the program, it is a fair assumption to consider that part of the signal as stationary. The fairness of the ergodicity assumption is ensured by using the unbiased estimate of the autocorrelation, i.e., the expected value of estimate is equal to the statistical autocorrelation and the

variance of the estimate asymptotically diminishes [84]. Such estimate of autocorrelation can reveal periodicities in a stochastic signal, even when they are hidden within non-repetitive patterns. The unbiased estimate of autocorrelation sequence of the performance signal $x(n)$ consisting of M samples is denoted by $\gamma_{xx}(l)$ and is defined by:

$$\gamma_{xx}(l) = \frac{1}{M-l} \sum_{n=0}^{M-|l|-1} x(n)\,x(n-l), l = 0, 1, \ldots, M-1 \tag{5.9}$$

where the index l is the time-shift (lag) parameter. The amplitude of the autocorrelation sequence at each l is proportional to the extent to which $x(n)$ and $x(n-l)$ match at that point. If both signals match, a peak will be seen at that lag, emphasizing the periodicity of the signal. The distance between peaks can be used to determine the period (i.e., time between two similar segments) of the signal. The amplitude of the peaks emphasizes the strong or weak periodicities in the signal. The estimate of autocorrelation attains its maximum value at $l = 0$, where the signal matches with itself perfectly.

The unbiased estimate of autocorrelation is not free from its constraints. As we will notice in the examples, the variance starts to increase at larger lags, i.e., when $l \rightarrow M$. The reason is the scaling factor in the definition of the unbiased estimate of autocorrelation given above. Therefore, the estimate of autocorrelation at large lags might have amplitude greater than $\gamma_{xx}(0)$ and thus will not be reliable. We will be working with the correlation coefficients only at smaller lags in the case studies of Chapter 6.

The shape, specifically the peaks, of the estimate of autocorrelation signal can help identify the loops in a program. The regularity or irregularity of these peaks confirms the data-independent or data-dependent control constructs used for the loops. Nested loops will usually show smaller peaks within the dominant peaks. If this observed phenomenon could be related to the program code, then the behavior of the loops can be completely specified. An *indicator sequence* will be introduced to determine the temporal location of loop iterations from the program for the case studies in Chapter 6 so as to validate the information obtained from the estimate of autocorrelation. Even without the availability of source code, the shape of the autocorrelation can characterize intricate communication

patterns in the time domain, such as for broadcast, which is difficult to decipher visually from a time-series plot.

5.2.3 Comparison of Temporal Patterns

Temporal patterns in performance signals obtained from different phases of the same program or different implementations of the same program can be compared visually. This comparison is often part of the performance optimization process for a program in which its performance needs to be assessed in contrast with alternative implementations. It is not always feasible or desirable to do this visually. A common technique that supports this is to estimate the cross-correlation of two signals, which emphasizes the degree of similarity between two different signals *w(n)* and *y(n)*. This technique is the time-domain analog of the detection of changes (i.e., degree of dissimilarity) between two performance images in the spatial domain.

The estimate of cross-correlation sequence is a generalized form of the estimate of autocorrelation sequence presented in the last section. The cross-correlation of two performance signals *w(n)* (consisting of N samples) and *y(n)* (consisting of L samples, where $N > L$) is obtained by shifting *y(n)* one step at a time and comparing it with *w(n)* (over all n) for all lags l to get the magnitude of the correlation coefficients at each lag. This operation can be represented mathematically by:

$$\gamma_{wy}(l) = \sum_{n=0}^{N-1} w(n)\, y(n-l), l = 0, 1, \ldots, N-1. \tag{5.10}$$

If the two signals are similar at a certain value of the lag index l, the magnitude of the cross-correlation sequence will have a peak at that lag. Again, the amplitude of the peaks is proportional to the extent of similarities between the two signals. It is useful to make the two signals as zero-mean processes (by subtracting the time-average of each signal from each of its samples), so that strong similarities (or dependencies) between the two show up as positive peaks and strong dissimilarities show up as negative peaks [62].

Estimate of cross-correlation can help evaluate the patterns of a program as "failed", "optimal", "desirable", etc. based on the *a priori* knowledge about the expected patterns in a program. Therefore, the estimate of cross-correlation can be used in two ways to identify the patterns:

1. To compare the template signal $y(n)$ with the performance signal $w(n)$ and to identify the temporal locations where strong similarities (dependences) between the two occur. Depending on the magnitude of the estimate of cross-correlation at those points, the likelihood of occurrence of the template pattern at that time can be determined visually from the plot.
2. To compare and distinguish performance signals obtained from different phases of the same program or different implementations of the same program.

5.2.4 Thresholding

Thresholding was defined for analyzing performance images. Thresholding is a useful generic function that can be applied to performance signals to isolate any patterns that depend on the amplitudes of the signal. A threshold is a linear operator that was denoted by *Th* for performance images and can be defined for a temporal domain performance signal $w(n)$ as:

$$Th(w(n)) = \begin{cases} 1, & \text{if } w(n) = th \\ 0, & \text{Otherwise} \end{cases} \tag{5.11}$$

where *th* is the numerical value of the threshold. The resulting signal is a binary signal that will be helpful to identify the occurrences of events that are related with the amplitude of the signal (i.e., the magnitude of a performance metric) during the execution of a program. The use of binary thresholding was found useful for analysis of particular types of temporal patterns in programs. However, the thresholding operation can be defined in any other suitable manner depending on the requirements of analysis.

A strategy for performance analysis in the temporal domain is represented by Figure 5-4, which shows two categories, as in the case of spatial domain analysis. The analysis steps starting from the performance data are similar to those in shown in Figure 5-1 for the spatial domain analysis. Performance data are transformed to the temporal domain

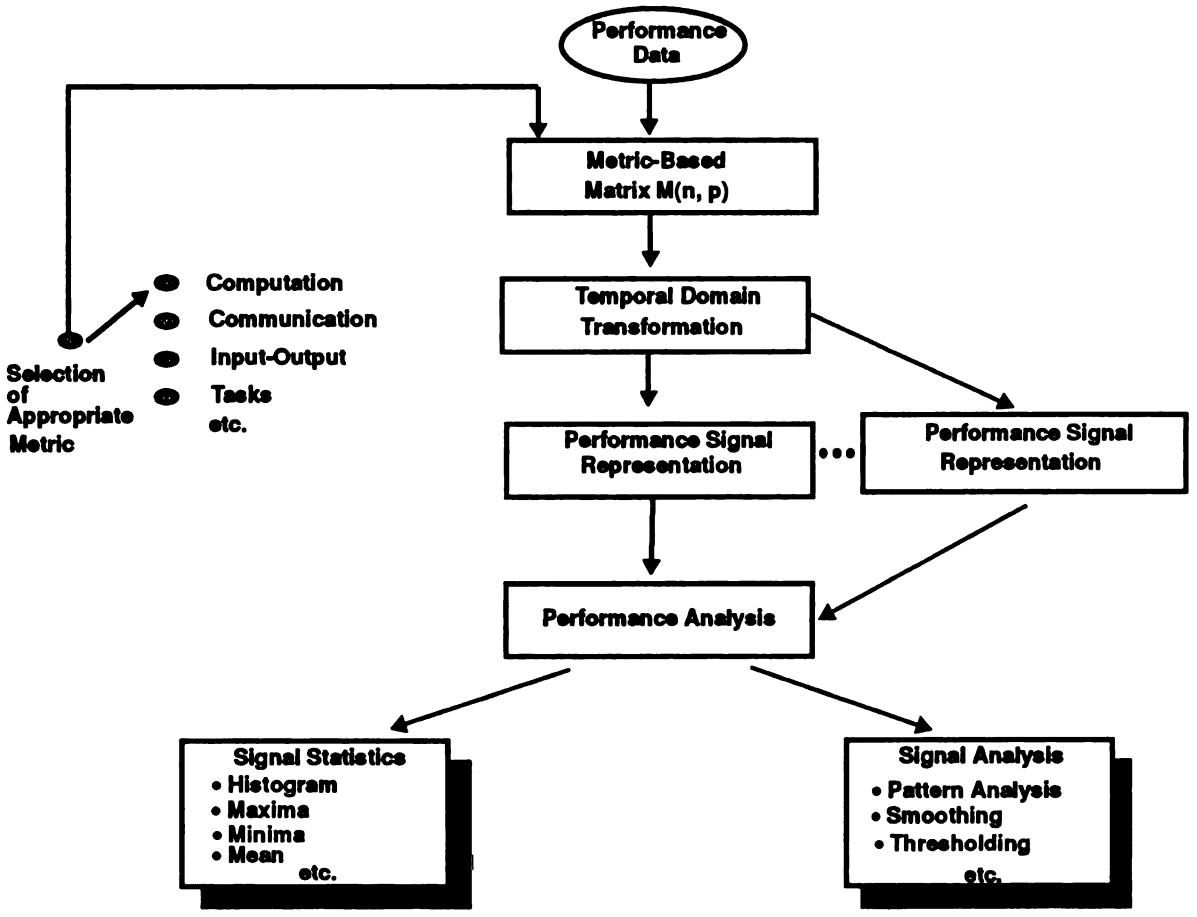


Figure 5-4. Application of signal processing to performance analysis in the temporal domain.

(through the event domain) to represent it as a discrete-time signal. Signal statistics can be obtained, but they are not as useful here as they were for performance images.

5.3 Analysis in the Event Domain

The event domain was also characterized by performance signals in Chapter 4. The event domain is primarily of interest to model the computation on a parallel or distributed system. Performance data (domain) consists of events that identify unique temporal and spatial states of the system [68]. However, the use of the event domain for developing models of program behavior is beyond the scope of this thesis, as it is a future direction of this work. We consider the present work as a basis for comparison with the results of any subsequent modeling work. Nevertheless, event domain performance signals can be used

to depict the dynamic behavior of the program, in addition to the temporal domain performance signals, with the "tolerance" that was precisely specified by theorem 4.3 in Chapter 4. State transition information can be used to develop Markov models for the program and system behavior, which were discussed in Chapter 2. We will conclude this section with an overview of the Markov modeling approaches that can used to analyze the program and system states in the event domain.

5.3.1 Representation of State Transitions

Event domain representation can be used to analyze the sequence of state changes during a program. Various system states are identified by an appropriate choice of performance metric. For instance, if the event domain performance signal $x(n)$ shows the system utilization values, then it can represent the "full utilization" by its maximum value (which is 1.0 as system utilization is a ratio), "no utilization" by its minimum value, etc. Only the transitions among various states are depicted, without representing transition-time between any pair of states.

Representation of state transitions is useful to understand the operation of the system while executing a particular phase of the program. For each phase of the program, it goes through a particular sequence of state transitions due to the various activities within that phase. An event domain performance signal represents this pattern with microscopic details and its analysis can be useful to understand the higher-level program abstractions.

5.3.2 Analysis of Trends and Patterns

An event domain signal $x(n)$ is a discrete-time (performance) signal, which is mathematically similar to the temporal domain performance signal $w(n)$. The only difference is that it does not contain information about the holding times of the states. Therefore, processing of this signal permits the analysis of higher-level abstractions, as in the case of temporal domain performance signal. We can use the same analysis methods that were described for temporal domain analysis, if we do not lose sight of the difference between $x(n)$ and $w(n)$.

A performance signal in the event domain can be smoothed to assess trends. In this case, smoothing is more important due to the absence of transition times that "hold" the value of the metric till the subsequent state transition. If this signal is smoothed using a moving averages filter (similar to the one used in temporal domain smoothing), then the average values of the metric can be represented during the execution of a set of instructions.

Higher-level programming abstractions and control constructs have their manifestations in the event domain. A particular phase of the program will cause a peculiar pattern of state changes that will appear in the event domain signal. These patterns can be identified depending on their repetitions or individual occurrences using estimate of autocorrelation. This is not different from what was done in the temporal domain and can be considered as a better approach to obtain the same results because the amount of storage space required for an event domain performance signal is much smaller compared to that for the temporal domain signal. Similarly, other pattern analysis techniques such as estimate of cross-correlation and thresholding are also equally suitable for event domain performance signals.

So far, we have discussed the application of the types of analyses that were performed in the temporal domain. From a practical standpoint, an event domain performance signal $x(n)$ requires less computing resources (such as cpu time, memory, floating point operations, etc.) than a temporal domain signal $w(n)$. This was the motivation behind theorems 4.3 and 4.4 given in Chapter 4 to establish the difference between the two signals from a purely DSP point of view.

5.3.3 Analysis of Event-Frequency Distribution

Statistical theory has led to the development of various (probability and frequency) distributions that are used in univariate data analysis. There is a duality between frequency distribution and probability distribution because statistical models of a phenomenon can be described in terms of either probability or frequency distributions of the data [26]. In the context of program performance analysis, we have access to the execution trace data in

the performance domain which is inherently multidimensional. We can transform this performance data to univariate data corresponding to a suitable performance metric of our choice using the transformations to multiple analysis domains defined in Chapter 4. We can use this univariate data to show the density function of the performance metric, specifying the relative frequency (or probability) of occurrence of each element of a discrete set of values. The graphical pattern of this function will have different meanings (e.g., event-frequency distribution, system load-balance, spatial clusters, etc.) depending on the context of the performance metric being used to find the frequency density function. The shape (or pattern) of this function can be used to model a whole program executed on a particular system by its statistical properties. These statistical patterns will be useful to judge the overall performance of a program.

State transitions due to the message-passing activity can be characterized by a histogram. This histogram shows the frequency-density function (from which the frequency distribution can be calculated as cumulative frequency histogram) over the range of system states (represented by the dynamic range of the performance metric). The event-frequency density function represents the number of occurrences of events contributing to each level of the performance metric (e.g., instantaneous processor utilization), which is divided into classes or clusters. A plot of the number of events along the vertical axis and the classes (or clusters) of metric values along the horizontal axis represents the event-frequency information for the performance metric, and is also known as a histogram. The extent of the horizontal axis is equal to the range of the metric, and that of the vertical axis, the maximum number of events belonging to a cluster. The area within the region pertaining to each cluster is proportional to the event-frequency of that cluster. If C_i is one of the L clusters of dynamic range of a metric, then the frequency-density function of the events is given by:

$$f(C) = size\{x(n) | (x(n) \in C)\} \tag{5.12}$$

for $i = 1,2,..., L$. The event frequency-density function $f(C)$ obtained this way shows the (relative) probabilities of events in the program belonging to particular ranges of the

metric. The event-frequency curve shows the gross characteristics of the event frequency-distribution. The curve can take on a particular shape that is distinctive for the program and which may resemble one of the standard distributions (such as normal, bimodal, skewed, etc.). This is useful information for performance characterization and optimization, especially if it can be extended to performance modeling by fitting a suitable theoretical distribution to this observed statistical data [26, 30].

The event-frequency distribution in the event-domain gives an overall idea of the dominant states of the program, in terms of transitions to those states. If the histogram is drawn from the temporal domain signal $w(n)$, it represents the frequency-density information about various system states (including state transitions and the states themselves). Such a histogram can help visualize the *load-balance* of the program. If the frequency curve is skewed to one side or another, it can provide an overall feel for the system load-balance or imbalance, provided that the metric being used represents system utilization.

5.3.4 Markov Modeling

Building up of realistic stochastic models of a physical phenomenon is a compromise between the degree of dependence among observed data values that the physical situation dictates and the degree of independence needed to facilitate probability calculations. The more independence built into a probability model, possibility of more explicit calculations become better. However, the realism of the model becomes more questionable. Markov processes frequently balance these two demands [88]. A Markov process has the property that the probabilistic structure of the future data value depends only on the present value, and not on the past values. Therefore, future states become conditionally independent of the past.

Markov chains are Markov processes with a discrete index set and a countable (or finite) state space. Program (or system) states can be defined and the transition probabilities can be determined from the event domain information to represent the program behavior as a

Markov chain [1]. Various states of the system during program execution can be represented in the matrix form and the transition probabilities to and from these states can be calculated from the performance data transformed to the event domain. Different states might represent different modes of system operations, such as user, system, I/O, idle, busy, etc. The transition probabilities calculated through Markov modeling can help analyze the probability of dominant trends of resource usage by a program. These transition probabilities can be represented graphically and can help analyze whether a given transition is *recurrent* or *transient* [88]. Steady state probabilities of various states can be computed and the accuracy of this analysis can be determined from the information obtained from other analysis domains by applying other types of analysis methods (presented in the previous sections) that do not depend heavily on the stochastic model for the system and program behavior.

Application of discrete-time Markov chains to model various aspects of parallel computer system behavior were presented in Chapter 2. Developing useful Markov models from the event domain representation is an important future direction of this work.

5.4 Significance of the Frequency Domain

Only one reference of using the frequency domain for program performance analysis purposes is known to us [1], but its advantages for understanding the behavior of a program are not yet well-known. Nevertheless, the frequency domain contributes valuable information to better understand the temporal domain behavior of the performance signal. Thus, a brief discussion of analysis using the frequency domain is presented in this section.

Performance signals, either in the time domain or the event domain, can show many variations, e.g., due to message-passing events occurring at a very frequent rate. In the time domain, there may be so many variations within a small interval of time that it is not possible to even distinguish them without adjusting the time scale. The event domain further compresses these variations. To observe the variations of the performance data, the

frequency domain is a natural choice, particularly when the data has already been transformed to a signal and DSP techniques are to be applied to it. The transformation of a signal to the frequency domain is the well-known discrete Fourier transformation (DFT), as given in the previous chapter.

The performance signal is not a periodic signal, thus it cannot be specified by a single frequency. It can be thought of as a combination of all frequencies in the range $0 - \pi$, which is the range that applies to real-valued, discrete-time signals [84]. We can consider the frequency $\omega = 0$ as the minimum frequency, which corresponds to no change in the signal. This is the case when the performance signal is flat, i.e., the performance metric remains constant throughout program execution. Conversely, $\omega = \pi$ corresponds to the maximum (sharpest) change in the signal and is practically limited by the resolution of the instrumentation timer. Changes having a particular frequency ω can occur at multiple instances during execution. This is reflected by the magnitude of the spectrum at that ω. If the spectrum shows a high concentration of energy at some frequency, it means there are many changes occurring at that frequency (or rate). This approach of analyzing the frequency of variations has been applied by Abrams, Doraswamy, and Mathur [1]. The frequency domain analysis that we apply consists mainly of filtering in the frequency domain (as opposed to the time domain, where it may not be as effective) to smooth a performance signal by removing its high frequency content to identify the trends in the performance data. The performance signal can be filtered through a digital low pass filter [84] whose design is specified by the DFT of its impulse response $h(n)$, given (ideally) by:

$$H(K) = \begin{cases} 1, & for \quad 0 \le K \le (M-1)/\alpha \\ 0, & for \quad (M-1)/\alpha \le K \le M-1 \end{cases} \tag{5.13}$$

It rejects the upper (1 - (M-1)/α)% of the frequencies and retains the lower ((M-1)/α)% frequencies in its spectrum. The spectrum of the resulting filtered performance signal can be specified as:

$$Y(K) = X(K)H(K), (0 \le K \le M-1) \tag{5.14}$$

and the output of the filter is transformed back to the time domain by applying an inverse discrete Fourier transformation (IDFT) to $Y(K)$.

Another indirect use of the frequency domain is for resampling the performance signals at a lower rate (decimation). We need to use an appropriate low-pass filter before the performance signal is resampled at a lower rate in order to avoid aliasing [84]. If the sampling rate is reduced by a factor of α, then the spectrum of the signal will have to be restricted to $(M-1)/\alpha$, as shown above by $H(K)$. The removal of the high frequency contents from the spectrum will impose smoothing on the signal which is desirable to analyze the trends in the performance signal.

5.5 An Illustrative Example

A simple example of the use of matrices to model performance data and transform the data to the analysis domains is presented here. This example will help to unify the ideas presented so far. For this purpose, a part of a linear system solver was instrumented. The program solves a 16×16 matrix on 4 processors of an nCUBE-2, which is a hypercube multicomputer. The program uses Gaussian elimination (GE) technique on the matrix, with the rows of the matrix distributed to the processors. Each processor owns a block of 4 rows (known as the (BLOCK,*) data distribution). All the processors execute the same instructions on their local data, depending on the rows of the matrix that they own (single-program, multiple-data (SPMD) programming paradigm).

The part of the program that was instrumented for this example is concerned with the "pivoting" operation. A broadcast is initiated by the processor containing the row being used for the current iteration to inform all the processors of the location of the column containing the maximum (pivot) value. This operation is later used to implement an exchange of columns at each processor to complete the pivoting. The instrumented code is given in Figure 5-5 for reference. The call to *tracenode* was used to activate the tracing of the subsequent part of the code upto the *traceexit* call.

```
/* G.E. Main loop */
  tracenode(500000,0,1);
  for (curr_iter=0; curr_iter<N; curr_iter++) {
  maxloc = curr_iter;
  r_indx = find_r_indx(curr_iter);
  if (r_indx > FLAG) {
  local_max(r_indx,curr_iter,&maxloc,&maxval);
  nbroadcast(&maxloc,sizeof(int),inode,mtype1,FLAG);
          }
  else {
  src = curr_iter / ROW;
  nbroadcast(&maxloc,sizeof(int),src,mtype1,FLAG);
          }
  col_exchange(decomp,curr_iter,maxloc,inode);
  traceexit();
.....

  }
```

Figure 5-5. Instrumented code for the example.

The loop shown in the segment of the code repeats for all the rows, but tracing is stopped by *traceexit* after the first call to *nbroadcast.* Execution proceeds normally, and the program terminates. At the end of the program, a trace file is generated from the trace data collected at each node. The trace file contains the information about each message-passing event (such as event-identifier, time-stamp of the event, processor sending or receiving the message, message size, etc.) comprising the broadcast operation. The events are not globally sorted with respect to their time-stamps due to distributed nature of the data logging, and the first preprocessing step is to sort the events according to the time-stamps, keeping the causality among the events intact. The (PICL) trace file after this sorting operation is shown in the Figure 5-6 for illustration purposes.

The first field in all the event records is an event-identifier to identify various types of events such as starting of trace at a processor, message send, message receive, computations statistics of a processor at certain time, time when tracing was stopped at a particular processor, etc. The second and third fields are used for the time-stamps, where the second field gives the number of seconds since the start of tracing and the third field

1	0	11	0	4	4	0			
1	0	11	1	4	4	0			
1	0	11	2	4	4	0			
1	0	11	3	4	4	0			
20	0	56	1	-2	17	0			
20	0	56	2	-2	17	0			
20	0	56	3	-2	17	0			
7	0	106	1	17					
7	0	106	2	17					
7	0	106	3	17					
11	0	106	1	0	0				
11	0	106	2	0	0				
11	0	106	3	0	0				
20	0	149	0	-2	17	0			
4	0	204	0	1	17	4			
11	0	204	0	0	0				
11	0	325	0	0	121				
4	0	367	0	2	17	4			
11	0	367	0	0	121				
8	0	399	1	0	17 4				
11	0	399	1	0	292				
4	0	455	1	3	17 4				
11	0	455	1	0	292				
11	0	488	0	0	242				
21	0	532	0	-2	17	0			
8	0	553	2	0	17	4			
11	0	553	0	0	242				
11	0	553	2	0	446				
12	0	553	0	0	0	2	8	0	
19	0	553	0	300					
11	0	577	1	0	414				
21	0	614	2	-2	17	0			
21	0	621	1	-2	17	0			
11	0	634	2	0	446				
12	0	634	2	1	4	0	0	0	
19	0	634	2	252					
11	0	641	1	0	414				
12	0	641	1	1	4	1	4	0	
19	0	641	1	316					
8	0	643	3	1	17 4				
11	0	643	3	0	536				
21	0	703	3	-2	17 0				
11	0	724	3	0	536				
12	0	724	3	1	4	0	0	0	
19	0	724	3	252					
13	0	106133	3						
13	0	120785	2						
13	0	135533	1						
13	0	237552	0						

Event type 1: *trace_start*
Event type 4: *send*
Event type 7: *recv_blocking*
Event type 8: *recv_waking*

Event type 11: *compstats*
Event type 13: *close*
Event type 19: *trace_exit*
Event type 20 & 21: *block_begin & end*

Figure 5-6. Format of a PICL tracefile and various event types for the example instrumented code.

gives the number of microseconds. Remaining fields are dependent on the event-type and the semantics used by the instrumentation system [45].

The PICL trace file can be visualized by using a conventional *space-time* diagram from Paragraph. The space-time diagram in Figure 5-7 shows the local computation and communication patterns. The processors (i.e., spatial locations) are listed along the vertical axis, while the time is along the horizontal axis. The broadcast is initiated by

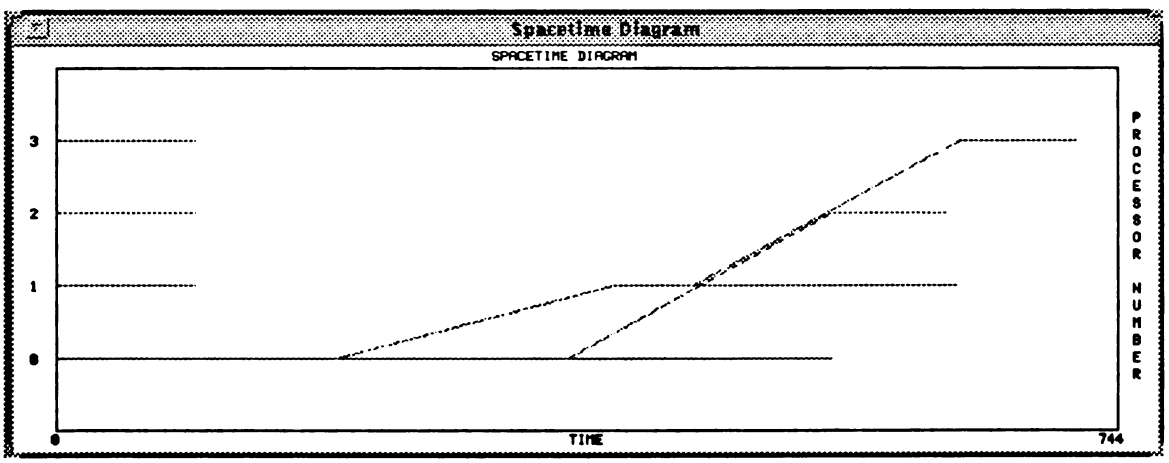


Figure 5-7. Space-time dynamics of broadcast.

processor 0, and a message is first passed to processor 1. This is indicated by a diagonal line from processor 0 to 1. Until processor 1 receives the message, it remains idle, which is indicated by discontinuing the horizontal line for processor 1. Then processor 0 sends message to processor 2; and processor 1 sends the same message to processor 3. At the end of this step, the message has been broadcast to all the processors, and the instrumented portion of the code ends.

The trace file being used for this example can be modeled by the *trace data* matrix T, represented in Figure 5-8. The rows of T containing * in a column indicate redundancy of this representation for making the matrix notation possible. The first column lists the event-identifiers of an event represented by a row. The second and third columns show the time-stamp, and the fourth column shows the processor number where the event was recorded. The rest of the columns are event-specific entries. For example, the third row

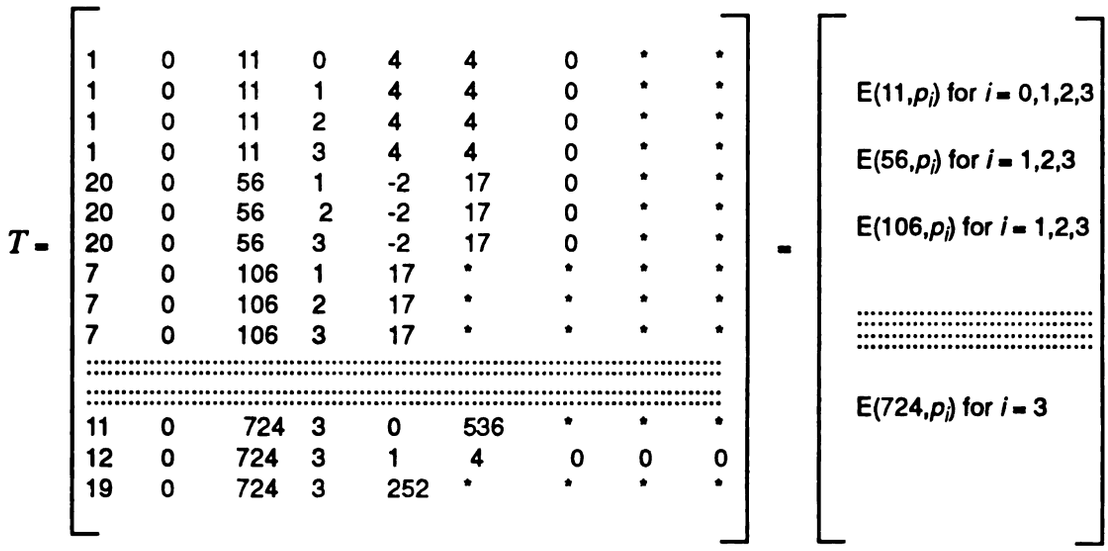


Figure 5-8. Trace data matrix for the example program.

from the bottom of T has an event-identifier 11. This event-record is known as a *compstat record* according to the semantics given by PICL [45]. Usually this is recorded at the end of a send or receive of a message by a processor. The second and third columns show that this record was logged after 724 microseconds from the time tracing was started. This record was logged by processor 3, as shown by the fourth column. This event record follows the receipt of the message by processor 3 at the end of the broadcast. The fifth and sixth columns indicate the amount of cumulative time that processor 3 has been idle (i.e., sending or receiving messages) upto the current time (724). It shows that processor 3 has been idle for 536 microseconds out of the total 724 microseconds of instrumented execution time. The block matrix on the right shows the same matrix with blocks of *concurrent events*, as defined previously.

The matrix T has so many dimensions of data that it is not directly useful for analysis or visualization purposes. Therefore we will not use this matrix as-is for further analysis. The analysis is based on the metric-based trace data matrix M from the trace data (not on matrix T). We have to select a metric for the purpose of preprocessing the trace data to get the metric-based representation as matrix M. We choose system utilization, which is the ratio of the processors that are busy to the total number of processors in the system. For

this purpose, we browse the matrix T (actually the trace file) with the help of a program and look for the event-identifiers that indicate a change in the status of a processor, i.e., from busy to idle or idle to busy. The state vector $v(n)$ holds the current status of all the processors at all n, as we browse down the rows of matrix T (i.e., for each time-stamp given by a row). The state vector is updated by the events causing a change in the status of any of the processors. The system utilization value is calculated at this point and a row is added in matrix M containing time, spatial location (processor number) and the current status of the processor (busy indicated by 1 and idle indicated by 0). This process continues till the end of the matrix T. The metric-based trace data matrix M for the above given T matrix is shown in Figure 5-9. This is one example of a metric-based matrix that

$$M = \begin{bmatrix} 11 & 0 & 1 \\ 11 & 1 & 1 \\ 11 & 2 & 1 \\ 11 & 3 & 1 \\ 106 & 1 & 0 \\ 106 & 2 & 0 \\ 106 & 3 & 0 \\ 204 & 0 & 0 \\ 325 & 0 & 1 \\ 367 & 0 & 0 \\ 399 & 1 & 1 \\ 455 & 1 & 0 \\ 488 & 0 & 1 \\ 553 & 2 & 1 \\ 553 & 0 & 0 \\ 577 & 1 & 1 \\ 634 & 2 & 0 \\ 641 & 1 & 0 \\ 643 & 3 & 1 \\ 724 & 3 & 0 \end{bmatrix} = \begin{bmatrix} M(11,p_j) \text{ for } j = 0,1,2,3 \\ M(106,p_j) \text{ for } j = 1,2,3 \\ M(204,p_j) \text{ for } j = 0 \\ M(325,p_j) \text{ for } j = 0 \\ M(364,p_j) \text{ for } j = 0 \\ M(399,p_j) \text{ for } j = 1 \\ M(455,p_j) \text{ for } j = 1 \\ M(488,p_j) \text{ for } j = 0 \\ M(553,p_j) \text{ for } j = 0,2 \\ M(577,p_j) \text{ for } j = 1 \\ M(634,p_j) \text{ for } j = 2 \\ M(641,p_j) \text{ for } j = 1 \\ M(643,p_j) \text{ for } j = 3 \\ M(724,p_j) \text{ for } j = 3 \end{bmatrix} = [n_i\, p_j\, m(n_i,p_j)]$$

where $n_i \in \{0,1,...,724\}$

$j \in \{0,1,2,3\}$

$m \in \{0,1\}$

Figure 5-9. Metric-based data matrix for the example program.

can be extracted from the trace data. This matrix is useful for obtaining performance images that depict the global state of the system at an instant of time, within the time scale defined by n_i. If we were interested in looking at the performance images that show cumulative utilization of the system, the third column would have listed the cumulative

busy time of the processors. Use of a state vector for this processing suggests that it can be accomplished in real time.

5.5.1 Transformation to the Spatial Domain

Now we show an example of drawing a performance image by transforming M to the spatial domain by selecting an observation time n_o. Suppose that observation time is set at 106 microseconds after the tracing was started, i.e., $n_o = 106$. While preprocessing the trace file, we know the total number of processors (P) that were used for the program. In this case $P = 4$. Therefore, we keep track of the states of four processors by using state vector $v(n)$. Processor states might be busy ($m = 1$) or idle ($m = 0$). This will continue till the end of the block $M(106,p_j)$ that includes all the concurrent events for time $n = 106$. As shown by matrix M, the four concurrent events at time $n = 11$ make all the processors busy (indicated by 1 in the third column). Then at the end of the concurrent events at $n = 106$, processors 1, 2 and 3 become idle whereas the (busy) state of processor 0 is unaltered. The state vector at this instant of time, $v(106)$, contains this information. Here we stop any further browsing through the matrix M. The spatial domain transformation at this time is given by:

$$T_1(M(n,p))|_{n=106} = S(p)|_{n=106} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 2 & 0 \\ 3 & 0 \end{bmatrix}$$

As indicated by equation (4.1) in the Chapter 4, the first column lists the number of processors while the second column lists the status of that processor (performance metric) and is identical to the state vector at this time. The matrix $S(p)$ undergoes the rendering operation to generate the performance image matrix $G(n_o)$ which in this case is given by:

$$G(106) = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

The first row indicates the metric values for processors 0 and 1. The second row is for processors 2 and 3. This matrix can be transformed into a digital image with image

enhancement operations, such as contrast stretching and pixel replication, to facilitate its viewing. This digital image is graphically drawn in Figure 5-10 to illustrate the idea. The values of the performance metric are mapped to a gray-scale (for this example, otherwise a color table is used), such that 1 is mapped to white and 0 is mapped to dark gray.

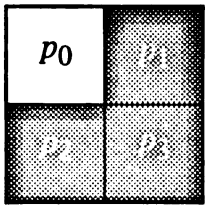


Figure 5-10. Performance image example.

5.5.2 Transformation to the Event Domain

The event-domain transformation is used as an intermediate step for transforming to the temporal domain, due to its practical advantages. Therefore, an example of transformation of the metric-based performance data matrix M to the event domain is shown here. According to the event domain transformation, we have:

$$T_2(M(n,p)) = M_e(n) = \begin{bmatrix} 0 & x(0) \\ 11 & x(11) \\ 106 & x(106) \\ \cdots & \cdots \\ \cdots & \cdots \\ 724 & x(724) \end{bmatrix}$$

Once again, the performance metric will be system utilization, and the performance signal given by the values in the second column of the above matrix represents changes in system utilization values. If $x(n_i)$ represents the fraction of the system being utilized (busy) at time $n_i = 106$, which is calculated with the help of state vector $v(n)$ at $n = 106$, then it can be obtained from M as:

$$x(106) = \frac{1}{4}[1\ 1\ 1\ 1]v(106)$$

$$= \frac{1}{4}[1\ 1\ 1\ 1]\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$= \frac{1}{4}\sum_{k=0}^{3} v(106) \quad = \quad 25\ \%$$

It should be noted that $x(0)$ is 100% with this metric, as all processors are assumed to be computing locally (i.e., busy). Also, the index of the vector x is not time in this case; it is the sample number of the performance signal. The performance signal $x(n)$ thus obtained from $M_e(n)$ is shown in Figure 5-11. This event domain performance signal is shown

$$M_e(n) = \begin{bmatrix} 0 & 100 \\ 11 & 100 \\ 106 & 25 \\ 204 & 0 \\ 325 & 25 \\ 367 & 0 \\ 399 & 25 \\ 455 & 0 \\ 488 & 25 \\ 553 & 25 \\ 577 & 50 \\ 634 & 25 \\ 641 & 0 \\ 643 & 25 \\ 724 & 0 \end{bmatrix}_{15 \times 2} \quad \text{where} \quad x(n) = \begin{bmatrix} 100 \\ 100 \\ 25 \\ 0 \\ 25 \\ 0 \\ 25 \\ 0 \\ 25 \\ 25 \\ 50 \\ 25 \\ 0 \\ 25 \\ 0 \end{bmatrix}_{15 \times 1}$$

Figure 5-11. Calculation of event domain performance signal for the example program.

(using Matlab) in Figure 5-12. As described above, it shows initial 100% utilization and then the system utilization varies according to the events during the execution of the instrumented part of the program.

5.5.3 Transformation to the Temporal Domain

After transforming the metric-based matrix M to the event-domain, it can be transformed to the temporal domain. We obtain a temporal domain performance signal $w(n)$ according to the equation (4.8). It is given as:

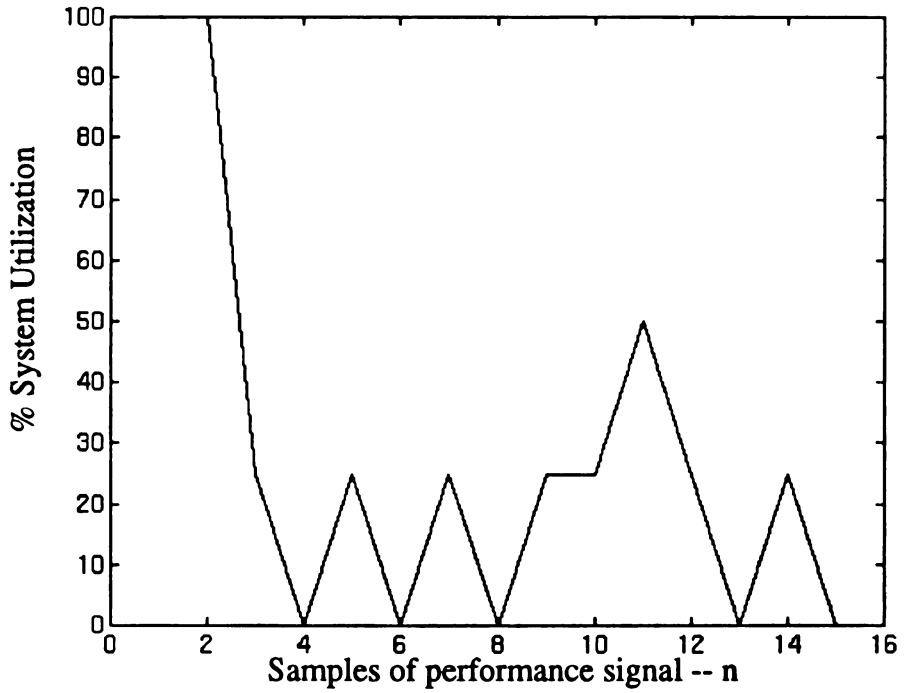


Figure 5-12. Performance signal in event domain $x(n)$.

$$T_3(M_e(n)) = w(n) = \begin{bmatrix} A(0) \\ A(11) \\ A(106) \\ \cdots \\ \cdots \\ A(724) \end{bmatrix}_{725 \times 1}$$

Again, the block A(0) is needed for the purpose of starting the time-scale from time $n = 0$, when tracing began. Since the whole system is being utilized at the start of tracing (100% utilization), the values of this block can be given as:

$$A(0) = \begin{bmatrix} 100 \\ 100 \\ \cdots \\ \cdots \\ 100 \end{bmatrix}_{11 \times 1}$$

The temporal domain performance signal is plotted (using Matlab) and is given in Figure 5-13. It shows all the state holding times that were missing in the event domain performance signal.

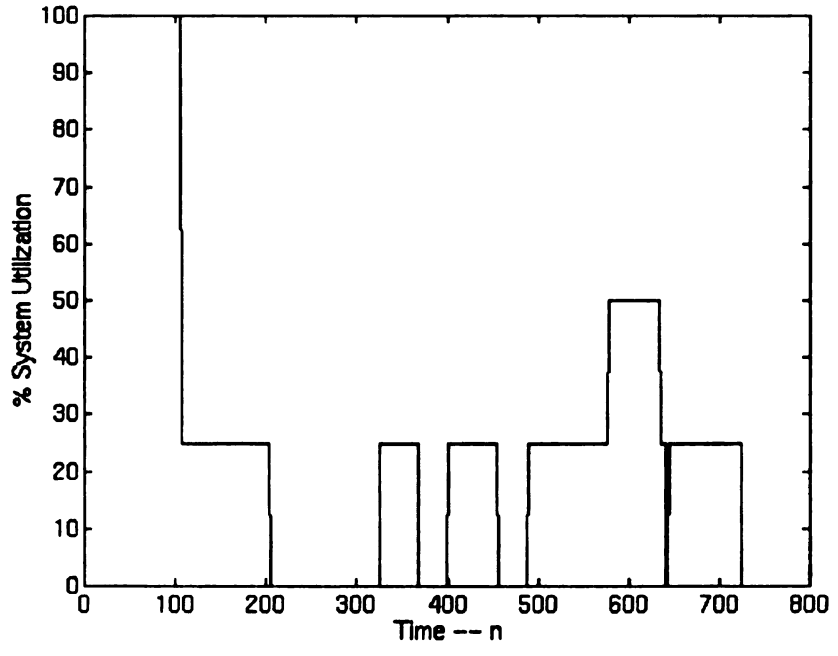


Figure 5-13. Temporal domain performance signal $w(n)$.

5.5.4 Transformation to the Frequency Domain

The frequency domain graph shows the spectrum of the signal. The spectrum of a temporal domain performance signal $w(n)$ can be obtained by applying DFT on $w(n)$. This is shown in Figure 5-14.

The spectrum can also be obtained from the event domain signal, as shown in Figure 5-15. The visual comparison between the spectra shows that the temporal domain signal has more energy at lower frequencies. This is expected due to the state holding times included in the signal.

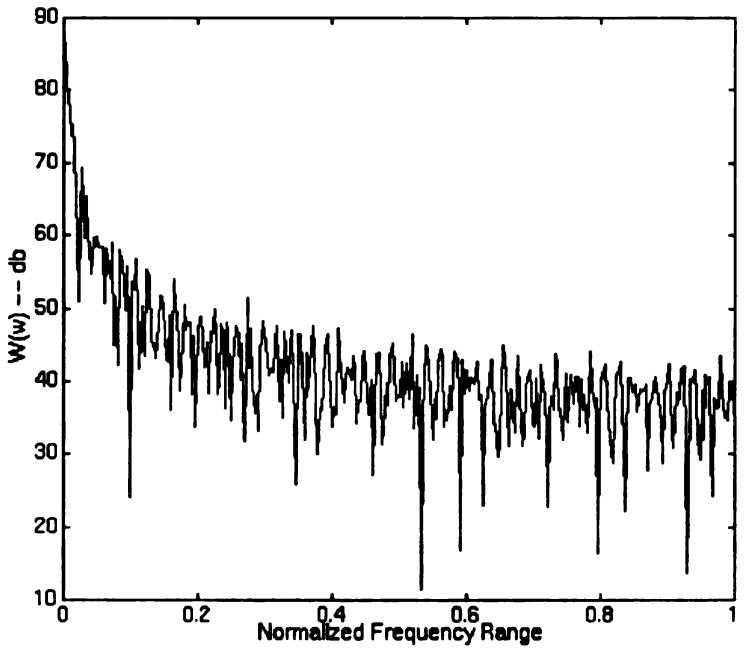


Figure 5-14. Frequency domain spectrum of the temporal domain performance signal.

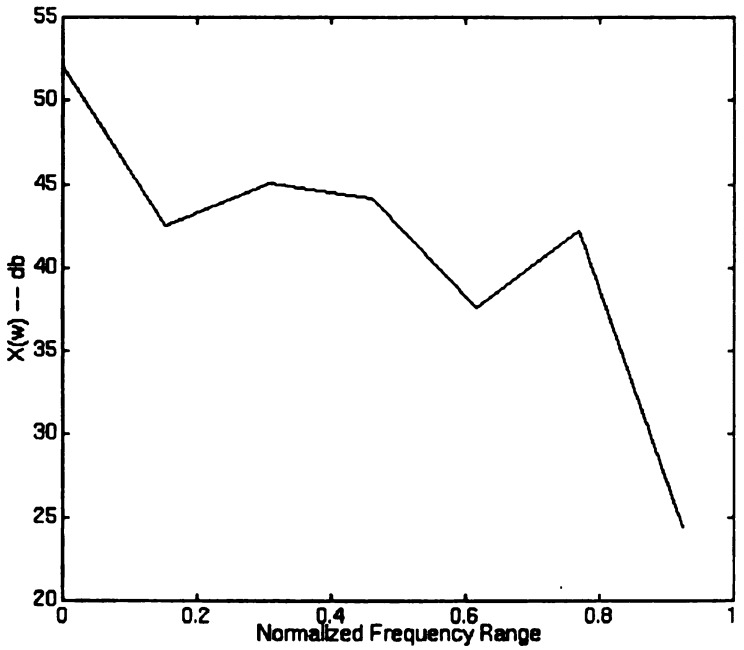


Figure 5-15. Spectrum of the event domain performance signal.

This concludes the example illustrating the transformation techniques defined earlier. The case studies presented in Chapter 6 are based on the same techniques presented by this simple example.

5.6 A Discussion on Applying Multiple-Domain Analysis

In this chapter, multiple-domain analysis methods have been presented independently of any other methods that are used for program performance analysis. The purpose of this section is to emphasize the fact that the multiple-domain analysis approach can be used in conjunction with typical, conventional analysis methods and displays in a systematic manner. This will ensure scalability, extensibility and efficacy of the program performance analysis in most of the cases. The purpose of the multiple-domain analysis approach is not to prove that the conventional methods and typical displays do not work in all the cases. These methods are effective in their own right. However, the use of multiple-domain methods can (1) supplement the user's knowledge obtained from conventional methods, and (2) provide analysis within the specific context of a domain. The advantages of the multiple-domain analysis approach to analyze program performance and behavior include:

- Identification of any macroscopic program execution patterns that might be present in the temporal, event or spatial domains.
- Identification and analysis of repetitive temporal patterns that might be linked with loops in a program.
- Analysis of macroscopic states of the system during the execution of a program in the spatial domain to represent control- and data-flow among the processors.
- Statistical characterization of a program by using the frequency distribution data of a metric.

The multiple-domain analysis methods are not free from limitations. Their main limitation is associated with the premise on which they were developed: they focus on one domain only at the expense of analyzing the performance data in more than one domain at the same time. Conventional tools usually allow this feature with specialized displays (such as the space-time diagram) which are very effective as long as the number of processors and volume of trace data remains small. Due to the advantages as well as limitations of

multiple-domain analysis, we suggest the following general guidelines for analyzing the performance of a particular program:

1. Conventional displays that represent both spatial and temporal dynamics of the program simultaneously should be used. This will be possible when scalability is not a problem and will contribute towards the user's familiarity of all aspects of the program.
2. Temporal domain information should be used to understand the temporal behavior and patterns of the program. The user's knowledge about the program is needed at this step to link the patterns with the program.
3. Spatial domain information can be used for more precise analysis of the system state at a give instant of time. This will complement the user's understanding of temporal behavior and patterns of the program.
4. Frequency distributions of the metrics should be used to statistically characterize and distinguish one program's performance from another.
5. Most of the statistical information obtained in each of the domains can be used to stochastically model the behavior of the program, which can be used to predict the values of a performance metric. Such models can be rarely built automatically and often require user input. Therefore, statistical information should be used carefully to avoid wrong predictions about the behavior of the program and system.

These are merely guidelines. We continue to refine them into a more well-defined methodology that a user can apply in practical situations. These guidelines assume that the user has access to conventional as well as generic data analysis tools to perform both conventional and multiple-domain analysis. We have proposed a toolkit approach [119, 120] in which conventional and generic tools are available to the user with a common data transfer protocol. Based on our experiences with conventional and generic data, signal, and image processing tools, we also propose that data can be transferred effectively among most of the tools if it is treated as a matrix [119].

In conclusion, the multiple-domain analysis approach is advocated in order to:

1. increase the scalability of the analysis and presentation methods;
2. precisely analyze a particular aspect of the performance in a particular analysis domain;
3. characterize the higher-level abstractions of the behavior of parallel programs from lower-level trace data; and
4. estimate and visualize various performance metrics in appropriate domains using rigorous analysis methods commonly used in statistical analysis, numerical analysis, and digital image and signal processing areas.

Chapter 6

Case Studies

In the previous chapters, we have rigorously defined the matrix representation of performance data and its transformations to the analysis domains. We have also identified the analysis methods in the analysis domains that can be applied for the purpose of more effective analysis of program performance. In this chapter, we will show the application of these methods to the performance analysis of some application programs executed on various parallel systems. In the first section, we explain the context of all the case study programs that will be discussed in the later sections, then we explain specific analysis methods in each subsequent section. We have not divided the case studies according to the analysis domains and analysis methods of that domain. Instead, our approach will be to present the application of these methods to accomplish various performance analysis objectives.

6.1 Context of Case Studies

We have chosen two types of programs to present the case studies, implemented on two types of parallel systems. First program approximates the value of *pi* (3.14...) by numerically evaluating the integrand $4.0 / (1 + x^2)$ on the interval [0.0, 1.0]. To compute the integral numerically, this interval is divided equally among P processors. Each processor further subdivides its region into m subintervals and computes and locally sums the area of its subintervals. After this local computation, all processors participate in a global sum-exchange operation of $\log_2 P$ exchange-add steps to finally get a precise value of *pi*. This is a simple program which will be used only for the purpose of introducing some case studies and more emphasis will be placed on a suite of second types of programs. These programs will be termed as linear solvers, that solve a system of linear equations by Gaussian elimination followed by backsubstitution. These programs were executed on an nCUBE-2 multicomputer [76] and were instrumented and visualized using

PICL and Paragraph toolset [44, 50]. The nCUBE-2 is a distributed-memory message-passing Multiple Instruction stream, Multiple Data stream (MIMD) computer. The Portable, Instrumented Communication Library (PICL) provides a tracing facility to record computation and communication events during program execution. It supports application-level monitoring by instrumenting the source code with calls to its communication library functions. The PICL routines are invoked during program execution by the events of interest and time-stamped tracing is accomplished concurrently on all processors. ParaGraph complements this monitoring by replaying PICL generated trace files and displaying a number of views.

Some of the case studies will involve another matrix solver benchmark known as *SLALOM* (Scalable, Language-independent Ames Laboratory One-minute Measurement). It is the first benchmark which compares performance of various computers on the basis of work that can be done within a fixed time (usually one minute). An in-depth discussion of the design of this benchmark is given in [48, 90]. Unlike many other benchmarks, it solves a real problem (optical radiosity on the interior of a box). The walls of the box are decomposed into patches (n). This number n is the figure of merit to rank various architectures instead of execution time, because the time is fixed and problem size scales according to the capabilities of the machine. *SLALOM* determines the largest problem size (i.e. n) that can be executed on that machine in less than or equal to one minute. There are seven tasks within *SLALOM* identified as *reader, region, setup1, setup2, setup3, solver,* and *storer.* We consider the *solver* task in this section. The *solver* performs *Gaussian elimination* and *back-substitution* methods to solve a system of linear equations arising from the radiosity problem. In the case studies using *SLALOM*, we analyze the operation of the *SLALOM* program (specifically *solver*) on a MasPar MP-1 data-parallel computer using 16,384 processing elements (PEs). The MasPar MP-1 is a distributed memory massively parallel Single Instruction stream, Multiple Data stream (SIMD) computer with a toroidal-mesh interconnection topology.

To prototype the application of multiple-domain analysis methods, we are using AVS [8, 9] to analyze and display performance images and the Signal Processing Toolbox of Matlab [71] to analyze and display performance signals. ParaGraph has been used when conventional displays are needed. Performance data are collected from program executions on actual machines and translated into AVS- and Matlab-readable formats, in terms of performance images and signals. We have translated PICL [44, 45] trace files from nCUBE-2 multicomputer executions and snapshot files [70] from MasPar MP-1 data-parallel computer executions.

A programmer assessing performance typically asks the questions, in order [95]:

- Is my program running correctly?
- How well is my program running?
- Can the performance be improved upon?

The second question is answered in part by a summary of the program's performance, such as that given by Table 6-1. It is the objective of *multiple-domain analysis* to provide insight into the second and especially the third questions by (1) focusing on particular domains of program execution where answers can be found and (2) offering analytical tools to explore the domains for answers. The remainder of this section describes the program, system, and metric used as the means of presentation.

6.1.1 The Program and System

In each case, the program under study is a variant of matrix solution by Gaussian elimination; and the performance metric is a form of processor utilization. These provide a familiar context only and do not favor the analysis methods. The parallel program consists of two main phases: (1) Gaussian elimination, where matrix data accesses occur in a pattern that successively moves "down" the matrix diagonal; and (2) back-substitution, where matrix data accesses occur in a pattern that successively moves "up" the matrix diagonal. Each phase consists of loops of repetitive operations. The programming model is SPMD (Single-Program Multiple-Data) for the nCUBE-2 and SIMD (Single-Instruction Multiple-Data) for the MasPar MP-1. A number of data distribution (or machine mapping)

strategies are possible. Using terminology from the Fortran D specification [36], the results presented here are from three strategies: (1) DISTRIBUTE (CYCLIC, CYCLIC); (2) DISTRIBUTE (BLOCK,*); and (3) DISTRIBUTE (CYCLIC,*). These are shown in Figure 6-1. All are two-dimensional distributions of the matrix among processors, where

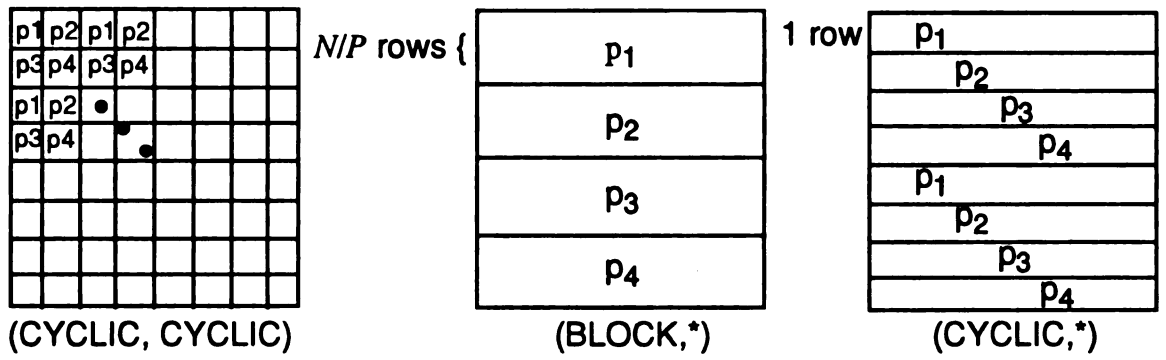


Figure 6-1. Data distributions (here, P = 4 and N = 8).

the latter two divide along the first dimension (rows) only. Assuming P processors and N rows in a decomposition, where N is divisible by P: the (CYCLIC, CYCLIC) distribution divides the decomposition in a round-robin fashion in the first and second dimensions (rows and columns); the (BLOCK,*) distribution divides the decomposition into contiguous blocks of size N/P in the first dimension (rows), assigning one block of rows per processor; and the (CYCLIC,*) distribution divides the decomposition in a round-robin fashion in the first dimension (rows), assigning every P-th row to the same processor. There are a number of opportunities for comparative analysis. With regard to the program, we can look at different phases, data distributions, algorithms, and problem (matrix) sizes. We can also use different system (machine) sizes. Each of these impacts the observed performance and will be used to demonstrate the analysis methods.

6.1.2 Performance Metric and Result Summary

The basic performance metric that will be shown in the following sections is *processor utilization.* This metric is obviously an important indicator of how well a program is running and using system resources. It also can be shown in some form in each of the

analysis domains. The term "busy" is often used to describe processor utilization, e.g., as the "number of processors that are busy" or the "percent of time that a processor is busy". In general, in a message-passing multicomputer, a processor can be computing, communicating, or idle (waiting to receive a message or done computing). If we only distinguish between "busy" and "idle", what does "busy" mean? We adopted the definition used by PICL: a processor is idle if it is executing system routines that would not be necessary if executed on a serial computer; otherwise, it is busy [45]. By this definition, a processor that is communicating is also considered idle. So, busy means doing only "useful computational work". While this is an important distinction in terms of characterizing program behavior, it has no impact whatsoever on use of analysis domains, and we mention it merely as part of the context in which results are presented.

Table 6-1 summarizes the program execution times for the different combinations (times are for programs without instrumentation). Note that considerably more time is spent in the Gaussian elimination phase than in back-substitution in each case. Also note that the execution times for the (BLOCK,*) distributions are lower than the corresponding (CYCLIC,*) distributions. Finally, these are fairly small runs in terms of machine sizes and problem sizes used. This was necessary to collect performance data having sufficient detail and keep the size of the trace files manageable. Alternative tracing strategies are needed for larger runs. However, the multiple-domain analysis approach is not impacted by the amount of performance data: it is scalable, given a scalable data collection facility.

TABLE 6-1. Summary of program execution times.

Distribution	Machine Size (P)	Problem Size (N)	Time: Gaussian Elimination Phase (sec)	Time: Back-substitution Phase (sec)	Total Execution Time (sec)
(BLOCK,*)	4	16	0.017436	0.002731	0.020167
(BLOCK,*)	16	64	0.210000	0.034843	0.244843
(BLOCK,*)	64	128	1.181063	0.174400	1.355463
(CYCLIC,*)	4	16	0.020850	0.004483	0.025333
(CYCLIC,*)	16	64	0.270537	0.053075	0.323612
(CYCLIC,*)	64	128	1.527301	0.259702	1.787003

Selected results that demonstrate the various aspects of our approach are given in subsequent sections.

6.2 Representing the Machine Mapping

An important consideration is how to present and analyze performance information in the context of the program without requiring completely specialized displays as for program visualization (or algorithm animation) in the application domain. At least, a tool should provide a dynamic display of the source-code. However, a practical alternative to application-specific displays is to represent the programming model [94, 95]. This is more general, yet gives the user useful information about the data and/or control structures of the program. Moreover, this likely is a lesser burden on the user because of the potential for compiler and/or run-time support of the model. For example, in the SPMD model, the relationship of performance data to program data and operations relies on the mapping of program data onto the machine (i.e., the data distribution or machine mapping). In fact, a *coupling property* can be identified that binds the performance display to the program. This property is similar to the relationship between processes and data illustrated in the data-parallelism view by LeBlanc et al. [65]. Use of this approach with the SPMD and data-parallel models is discussed in more detail in [95], however, an example is given here to show how performance images in the spatial domain can be related to the application domain.

6.2.1 Method: *Performance Images*

A one- or two-dimensional performance image can represent the machine mapping for structured problems by depicting the spatial relationship among processors with respect to data distribution. In general, if a so-called coupling property can be stated, any relationship among performance data also can be projected to effectively use performance images. This coupling property can be used while rendering the spatial domain matrix to an image, as discussed in Chapter 4.

6.2.2 Example: *Linear Solver*

Consider the case of the backsubstitution phase of the program on 16 processors of the nCUBE-2 using a (CYCLIC, CYCLIC) distribution. A coupling property of this mapping is that a row of processors "owns" a row of the matrix, and a column of processors "owns" a column of the matrix. A binary performance image is shown in Figure 6-2, in which each cell is encoded with the state of the processor at time n_o, where white denotes busy status (including communicating) and gray denotes idle status. The image is updated over time to reflect changes in state. This image, particularly when animated, clearly shows the sequential effect of stepping row by row up the matrix diagonal during the backsubstitution phase: on average, at most one row of processors in the grid is busy at any time. In one view, we can see the utilization of the system *and* the pattern of data movement in the program. This view helped to "decode"—in terms of matrix data movement—the more conventional space-time diagram shown adjacent to it. It also helped to optimize the data distribution strategy for this phase. In the recent release of ParaGraph, the Processor Status and Network displays show similar information as this type of performance image.

6.3 Viewing the Macroscopic State of Systems

The preceding section demonstrated the use of a performance image in the context of a small-scale system. Performance images and signals, however, are especially useful for representing (and possibly even modeling) the macroscopic state of a large-scale system in a particular domain. A macroscopic display is the first step in a top-down approach to performance analysis. Once again, let us consider the spatial domain and the machine mapping.

6.3.1 Method: *Performance Images*

It was shown in Chapter 5, that the performance images can be used to represent the state of the system at a particular observation time during the execution of the program. This

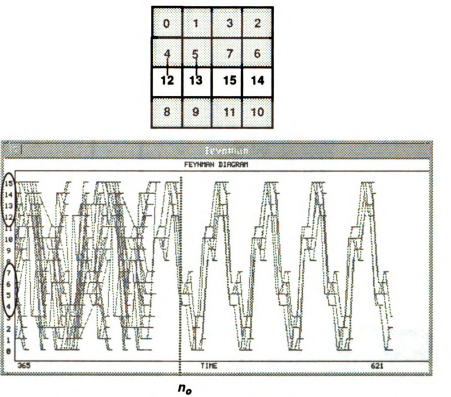


Figure 6-2. Performance image used to represent the machine mapping.

was made possible by mapping the value of performance metric (cumulative utilization, in the accompanying example) to a particular color, using the available color table index.

6.3.2 Example: *SLALOM*

Two cases of (symmetric) matrix solution on a 16,384-PE MasPar MP-1 (128 × 128 PE grid) using a (CYCLIC, CYCLIC) distribution are compared. In one case, there is an even distribution of matrix elements to PEs (1024 × 1024 matrix); in the other, an uneven distribution (1026 × 1026 matrix). The mapping for the uneven case is pictured in Figure 6-3. Performance images of final PE utilization (where each cell is encoded with the percent of time the PE was busy, or in the active set) are shown in Figure 6-4 (PE 0 is denoted by the upper, left cell). Two results that relate to the program are clearly visible in the images. In each image, the effect of a symmetric solution algorithm is shown by the contrast in utilization between the lower (greater than 50%) and upper (less than 50%)

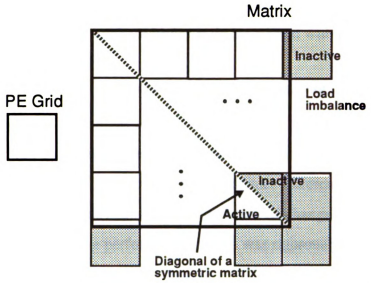


Figure 6-3. Machine mapping for uneven distribution.

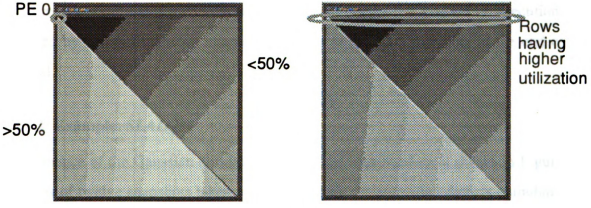


Figure 6-4. Performance images of PE utilization: even and uneven distributions.

triangles. In the second image, the effect of an uneven distribution is shown by the strip of PEs at the top having higher utilization. Notice that such views help to identify classes of behavior in a large system so that more detailed analyses—within a class, among classes, or of outliers—can be conducted.

6.4 Analysis by Small Multiples

Analysis by small multiple is a classical visualization technique for analyzing an image by incrementally increasing the time and analyzing the changes while keeping the

perspective fixed [116, 117]. This method can isolate the changes occurring over time. An application of analysis by small multiples is demonstrated with performance images.

6.4.1 Method: *Image Subtraction and Binary Thresholding*

The visual detection of changes over time can be difficult if the incremental changes are small or obscured by other effects. Change detection via image subtraction techniques is applied to successive performance images to enhance differences and similarities and to highlight the magnitude, direction, and distribution of change. The application of image subtraction technique to the performance images was rigorously defined in Chapter 5.

Image subtraction can be coupled with binary thresholding technique to focus only on the type of changes, i.e., positive or negative in terms of the cumulative utilization metric represented by colors. A positive change indicates the increase in the value of performance metric at a particular spatial location during the interval of execution time depicted by two performance images, one taken at the start and other at the end of this interval.

6.4.1.1 Example: *SLALOM*

Performance of the Gaussian elimination algorithm within *solver* is shown in Figure 6-5. A series of twelve snapshots taken over time during the algorithm depicts cumulative PE utilization. PE 0 corresponds to the upper left cell of the performance images, and PEs are arranged in row-wise order. A two-dimensional scattered decomposition method was used. The sharp contrast in utilization between upper and lower triangles of the PE grid is due to the behavior of the symmetric matrix solution method (discussed in detail in [93, 95]). These images are rendered with each of the 16K PEs occupying one pixel. The performance images of Figure 6-5 are quite simple, yet it is not trivial to extract meaningful information from the "raw" images. To reap the benefits of this representation for performance visualization of large-scale parallel systems, we need to apply analysis techniques.

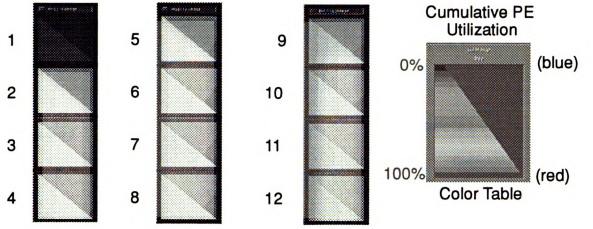


Figure 6-5. Performance images of GE in SLALOM (Series of 12 Snapshots Over Time).

Figure 6-6 shows the results of the subtraction (3 – 2) to detect the change in utilization from snapshot #2 to snapshot #3. In the image statistics box, the range of color indices is between –5 and 5 for the subtracted image, which indicates relatively small changes with respect to the full range. These indices are then mapped to the full color range in AVS so that the lower half of indices represents negative change and the upper half of indices, positive change. From the histogram for the subtracted image, you can observe that most PEs incur increases in utilization, and the statistics give a mean index of 2.9 (in the range –5 to 5). By inspecting the color image in conjunction with the color table, you can further see that relatively larger positive changes occur in the upper triangle. Therefore, by comparing successive performance images for a given program, the user can observe spatial changes for a given program.

6.4.1.2 Example: *SLALOM*

Consider matrix solution on the MasPar MP-1, for the uneven distribution case. Images were generated at a series of times throughout program execution. Two images can be subtracted to show the spatial utilization difference pattern from one time to the next, and a subtracted image can be further transformed into a simpler binary image via thresholding by mapping increases in utilization to one value (shown as black) and non-

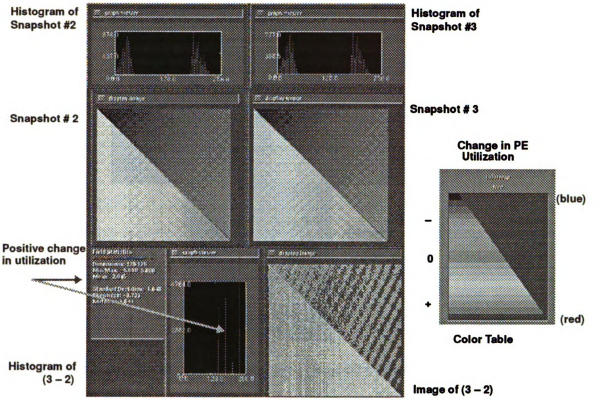


Figure 6-6. Change detection via image subtraction.

increases to a second value (shown as gray). The resulting images are shown in Figure 6-7. Note how the load imbalance in the top rows is emphasized.

6.4.2 Method: *Spatial Pattern Matching*

Many application programs exhibit specific patterns in spatial domain if performance images are used for characterization of the spatial domain. These patterns can depict the behavior of the program in spatial domain and can be used for comparison among various implementations or phases of the same program. This comparison can be carried out by using image processing technique of template matching. One image is used as template and is subtracted from the image under analysis to determine the degree of dissimilarities at all spatial locations, provided that two images are of same size. This method was explained in Chapter 5.

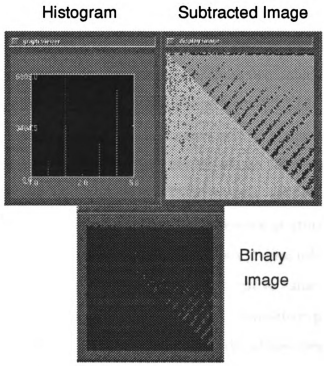


Figure 6-7. Analysis by small multiples using image subtraction and binary thresholding.

Template matching can be used to determine any changes in the performance metric (and hence the performance of a program) by comparing its different implementations. One performance image from one program can be used as reference (template) and other performance images from other implementations of the same program can be compared with the template. This procedure will result in identifying those spatial locations where metric showed maximum change, and can identify any bottlenecks in the performance of a program.

Spatial patterns will be discussed further after we analyze repetitive and non-repetitive patterns in temporal domains. It will be shown that the spatial patterns and their analysis techniques supplement the information obtained from temporal domain.

6.4.2.1 Example: *Comparison among Data Distributions*

In this example, we will compare the patterns from two implementations of GE phase of the linear solver implemented on 64 processors of nCUBE-2 multicomputer system. The

first implementation uses (BLOCK,*) data distribution and the performance images showing cumulative busy time at the end of this program will be used as template. Second implementation uses (CYCLIC,*) data distribution for the same GE phase on 64 processors. Figure 6-8 shows the comparison of these two implementations by subtracting the template image from the performance image from (CYCLIC,*) implementation. The subtracted image is shown below the two images with its histogram on its left side and the statistics. The red color at the top left corner of the subtracted image indicates that processors that owned top rows in (CYCLIC,*) case utilized those processors better compared to (BLOCK,*) implementation. The difference of utilization at other processors is not a great deal. However, this analysis also shows precise information by statistics and histogram. Histogram shows more occurrences at negative values showing the dominance of "negative" changes in utilization. i.e., overall cumulative processor utilization was better in the case of (BLOCK,*) implementation (by almost negligible margin of 1.2%) compared to that in (CYCLIC,*) implementation.

6.4.2.2 Example: *Comparison among Phases*

Template matching technique can be used for comparison of performance of various phases of a program. Such comparison can often help to precisely locate the part of the program that is deteriorating the performance of whole program. The linear solver is divided into two phases. First phase is to convert the matrix into a lower triangular form using GE and the second phase uses BS to calculate the solution vector. The performance image representing the cumulative processor utilization during BS phase can be used as a template image (for no particular reason, choice could be GE phase) and corresponding performance image from GE phase can be subtracted from it. The difference image is shown in Figure 6-9 with its histogram and statistics. Difference image shows negative values (represented by blue color) at the processors having top rows of the matrix i.e., the processor utilization was lesser for these processors during GE phase compared to that during BS phase. Similarly, processor utilization of processors that owned bottom rows of the matrix was better in GE compared to that for BS phase, therefore, the difference image

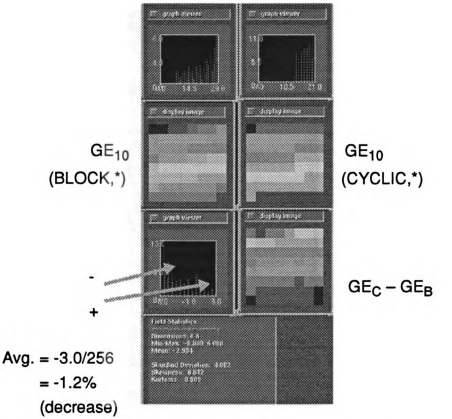


Figure 6-8. Comparison among GE phases of different data distributions via template matching.

show positive values (represented by red color) at those spatial locations. This comparison is consistent with our knowledge of the program according to which matrix data accesses moved "down the diagonal" in GE phase. Processors that owned bottom rows were more utilized, whereas matrix data accesses moved "up the diagonal" during BS phase, therefore, processors that owned top rows were more utilized. Histogram and statistics represent this information more precisely in the Figure 6-9 showing a comparatively better utilization (by 5.4%) in the case of GE phase.

6.5 Analysis of Temporal Patterns

Temporal patterns of parallel programs are either of repetitive nature or unique. Repetitive patterns are likely due to the loops in a program. Patterns can be unique due to the nature of the high-level message passing involved, or might even be defined arbitrarily by the

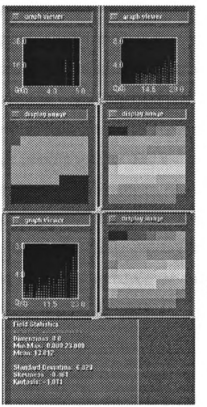


Figure 6-9. Comparison among GE and BS phases of linear solver.

user who is expecting them in his program. The reliable identification of such expected (or suspected, in the case of debugging) patterns can help understand the behavior of the program and be useful for optimizing program performance. The temporal program pattern analysis techniques presented here are basically to allow the user identify the possibilities of finding any expected pattern in the performance signal obtained from that program. These methods can be used in conjunction with the typical program visualization displays if the user is familiar with the patterns in that particular display. Visualization tools usually do not provide any support to the user in order to identify the known patterns, therefore, the *a priori* knowledge of the user about the behavior of a program is not utilized. Our main objective will be to utilize this *a priori* information from the user and by using appropriate techniques to identify the temporal locations during the execution of a program where these patterns are likely to be present.

6.5.1 Examples of Patterns Resulting from Message-Passing

Before analyzing the program patterns (repetitive or otherwise), we present some examples of spatial and temporal patterns resulting from high-level message-passing calls. These patterns will be demonstrated using space-time diagram of Paragraph [50] and (event domain) performance signals using Matlab [71]. This will allow us to combine the spatial and temporal dimensions inherent in these patterns and the interactions of the processors during the message passing activity. The use of performance signals will allow us to introduce them for representation of patterns, as our patterns analysis technique is based on this characterization. The objective is mainly to introduce the reader with the patterns resulting from the often-used message-passing calls using conventional displays as well as performance signals. We will introduce the patterns resulting from broadcast, global summation and barrier synchronization calls.

6.5.1.1 Broadcast

Broadcast is a high-level abstraction of inter-processor communication. Several algorithms use broadcast to distribute particular information from a source processor to all other processors in the allocated ensemble of processors. Our objective is to determine some specific patterns that broadcast might generate in temporal and spatial domains. This pattern will serve as a template pattern for a given instrumentation system and for a given number of processors. Every instrumentation system can handle broadcast in its own particular way, therefore, a template that works for one instrumentation system might not work for another due to the difference of communication protocols used. It might also depend on the type of interconnection network used, such as hypercube, mesh, etc. because various architectures can have different optimal protocols for implementing broadcast.

The broadcast shown by Figure 6-10 was implemented on 16 processors of nCUBE-2 multicomputer system using PICL's low-level primitives [45] to accomplish the broadcast of message originating from the processor 0. The diagonal lines in the space-time diagram show message-passing activity between processors listed along the vertical axis and

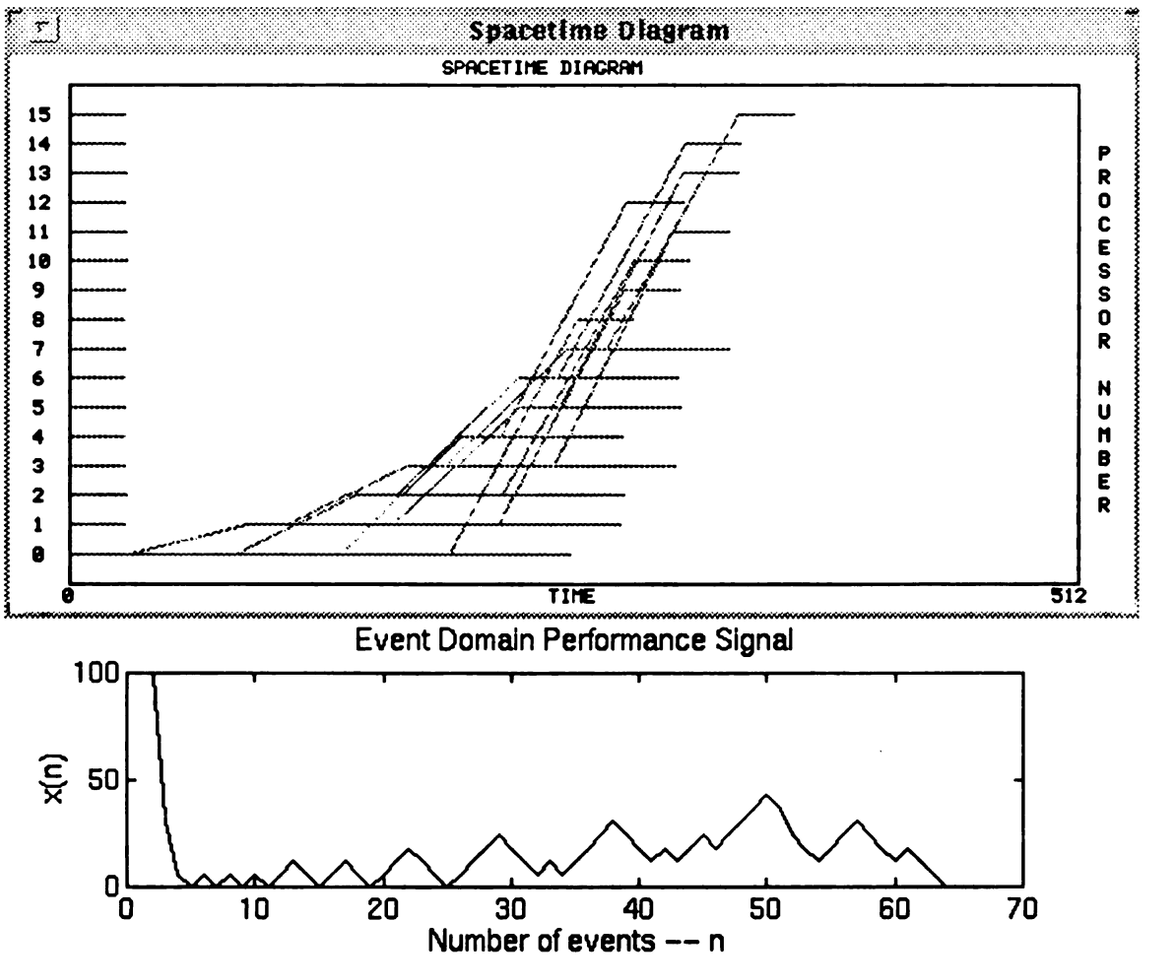


Figure 6-10. Spatial and temporal dynamics of Broadcast among 16 processors.

passage of time is shown along horizontal axis. The broadcast proceeds in such a way that each processor passes the message to its four neighbors, as there are 16 processors (4-dimensional cube). The performance signal $x(n)$ shows the system utilization (in percent) along vertical axis and number of discrete samples (or occurrences of events of interest where the metric changed) along horizontal axis. It is clear that the system utilization was very low when at the start of broadcast (the peak at n=0 is due to the synchronization call implemented by PICL, in order to generate consistent time-stamps) as most of the processors are waiting for the messages to be received. The system utilization periodically increases as the message reaches the processors and they exit the message receiving phase and become busy (according to PICL's definition). This is a specific pattern, both in time and space, and can be expected to occur in programs using PICL's broadcast call.

6.5.1.2 Global Summation

Global summation is one of the combining operations that PICL can provide at higher level, for summation of distributed vectors. The summation operation results in a single vector at the "root" node of this call which is a sum of a set of distributed vectors. The space-time pattern of global summation operation involving 16 processors is shown by Figure 6-11. The root node in this case is processor 0 which ends up with the global sum.

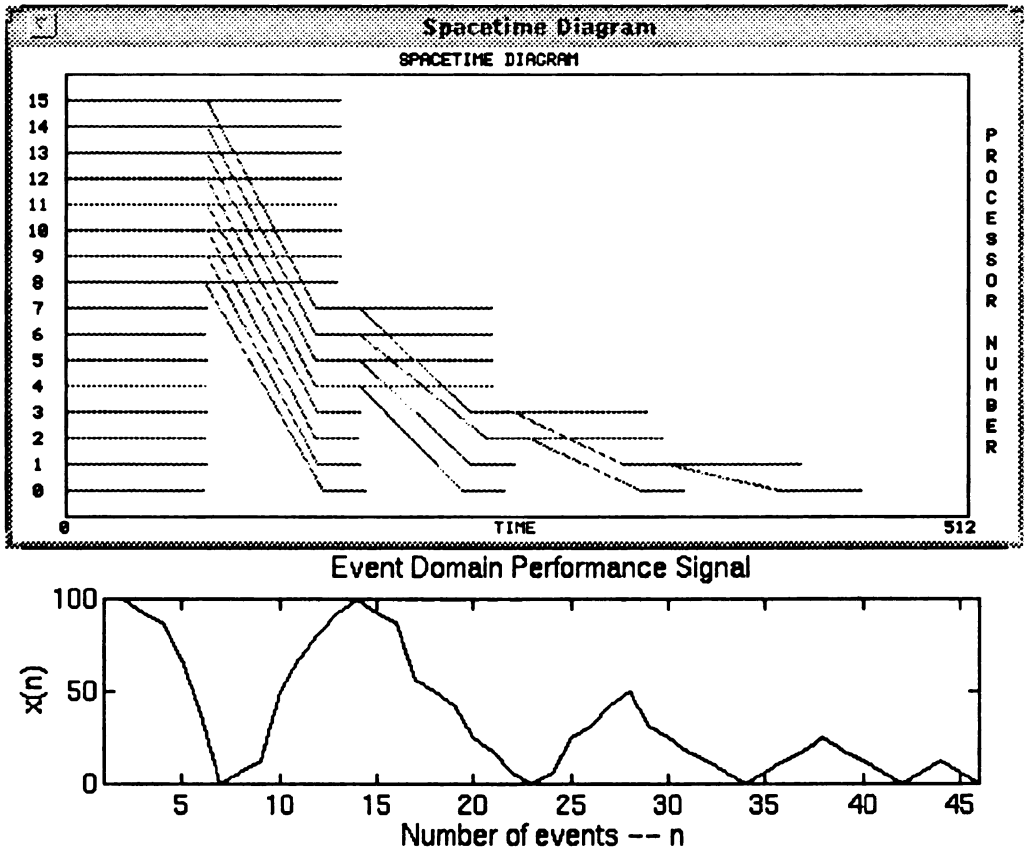


Figure 6-11. Spatial and temporal dynamics of global summation.

As shown by the space-time diagram summation process proceeds in a logarithmic fashion. First eight processors come up with their local sums and pass them along to other set of eight processors. Then in the next step four processors communicate their results to other four processors, and so on. The same activity in the temporal domain is represented by the performance signal $x(n)$. System utilization (in percent) is shown along the vertical axis. After the initial peak due to synchronization, the system utilization is very low as

half of the processors are passing messages (idle according to PICL's definition). System utilization then gradually starts decreasing in a periodic manner due to logarithmically smaller number of processors taking part in communication. This process repeats four times. Again this is a specific pattern and can be found in a program using global summation call.

6.5.1.3 Synchronization

Synchronization is a high-level communication process that can be used to synchronize the activity of processors in a distributed system. Such calls are important for the instrumentation purposes. PICL uses this call after opening communication channels among the processors to synchronize their clocks. This is needed to ensure consistency of the time-stamps of the distributed event traces. This call is implemented by having the processors exchange specific messages at the end of which their clocks become synchronized. The specific pattern of this call can be shown by the space-time diagram shown in Figure 6-12. After the processors are synchronized, the system utilization holds at its maximum value (100%) for certain amount of time, as shown by the performance signal. This phenomenon can be noticed in all the space-time diagrams and the performance signals that will be used for the analysis of patterns.

These examples of the spatial and temporal patterns show that various high-level message-passing activities generate peculiar spatial and temporal patterns. Knowledge about their shape in a particular domain (such as spatial or temporal) with their characterization (as performance images or signals) can prove valuable to identify them to debug or analyze the performance of a program. In the following, we present a hierarchy of techniques to analyze these patterns. For the purpose of analysis, we have classified both repetitive and unique patterns as *macroscopic* and *microscopic* patterns, and have treated them separately due to different degrees of precision required to analyze each type of these patterns.

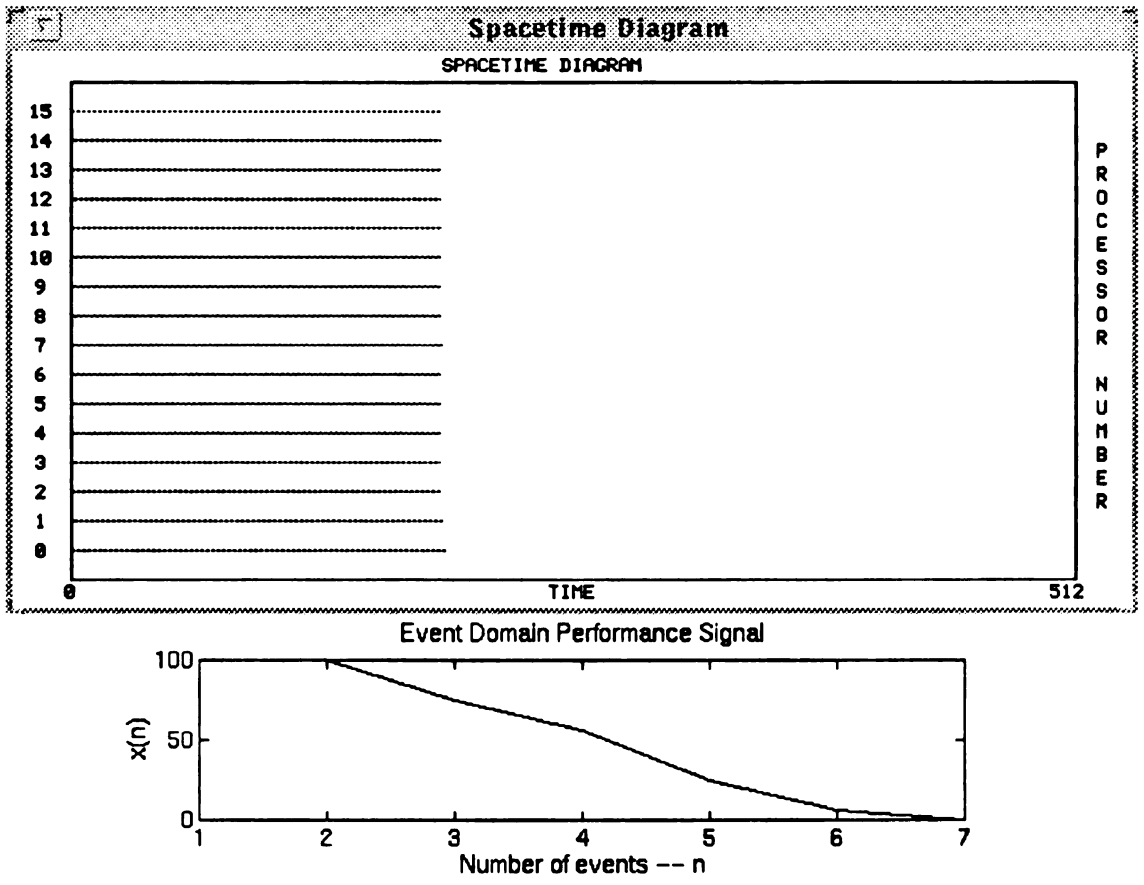


Figure 6-12. Spatial and temporal dynamics of synchronization.

6.5.2 Analysis of Macroscopic Patterns

In some cases, if the number of samples in a performance signal is not excessively large due to a large system and/or problem size, then macroscopic patterns and trends can be directly observed from the performance signal. As stated above, these patterns are often hidden due to a large amount of variance present in the signal. If we could remove this high variance from the performance signal (which is a time-series), the macroscopic patterns and trends (repetitive or otherwise) will immediately become clear. This is possible by applying appropriate DSP techniques to filter out the high variance contents of the performance signal which will result in a "smoothed" version of the performance signal. The method that we have selected applies a digital moving averages filter to smooth the performance signal. We give a brief introduction to this method and refer to [18, 42, 84, 85] for precise details of the implementation.

6.5.2.1 Method: *Moving Averages*

The use of moving averages for the removal of high variance from time-series performance data for the purpose of smoothing is a well-known technique. Researchers [17, 73] have shown its use for the identification of *stationary phases* of a program and have used this information for the prediction of speed-up characteristics [17]. The only difference in our technique is that we have characterized the time-series data as a discrete-time signal and have designed a digital filter for this purpose. This digital filter is a finite impulse response (FIR) type of filter and is causal; therefore, it can be implemented in real-time to calculate the moving averages sequence on-line. Design of this filter was specified in Chapter 5.

The order of the moving averages filter is data-dependent. If there is a very large number of fluctuations in the performance signal during a short time, generally a higher-order filter will be required. Disadvantages of using this filter include the loss of data at the starting point and the effect of the extreme values of the data. However, the filter adequately satisfies our objective of analyzing macroscopic patterns. We use other techniques for other purposes, such as thresholding for locating outliers in the signal (discussed in section 6.5.5.1). If there are any macroscopic repetitions of dominant patterns (for instance, changes in utilization due to repeated communication calls) they will show up in the smoothed signal. The following example demonstrates the use of moving averages to identify and analyze macroscopic repetitive patterns.

6.5.2.2 Example: *Linear Solver Program*

A linear solver program is a commonly used application program that solves a system of linear equations $Ax=b$ by using Gaussian elimination (GE) on the coefficient matrix A, followed by backsubstitution (BS) to determine the solution vector x. In order to show an example of the identification of macroscopic repetitive patterns, we have selected the BS phase of a solver of a linear system of dimension 64, executed on an nCUBE-2 [76] multicomputer using 16 processors. The coefficient matrix was distributed by using a (BLOCK,*) data distribution [36], where each processor owns a block of rows of the

matrix, and the matrix had already been converted to an upper-triangular matrix using GE. The space-time diagram for this phase is shown in Figure 6-13. There are several

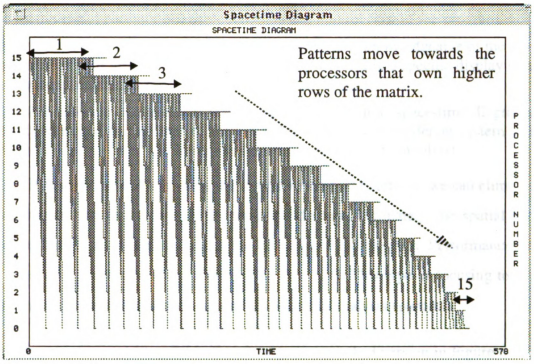


Figure 6-13. Space-time dynamics of backsubstitution algorithm (Paragraph).

interesting patterns related to the spatial and temporal activity of the system during the BS phase. Due to the type of data distribution used, each processor owns a block of four rows. The BS algorithm calculates one solution per row, starting from the bottom row of the upper-triangular matrix. At each step, the computation moves to the next higher row until all the solutions are calculated from all the rows. The space-time diagram shows a series of four distinct patterns due to message-passing involving one processor sending messages to all other processors owning higher rows of the matrix, starting from the processor that owns the bottom four rows (i.e., processor #15). This message-passing is needed to communicate the element of the solution vector that was calculated at one row to all the processors owning higher rows. We make the following comments on the patterns of this space-time diagram:

- The patterns are of a repetitive nature, with macroscopic patterns repeating fifteen times and each macroscopic pattern consisting of four similar "sub-patterns", due to each processor owning four rows of the matrix.
- Similar patterns shifting gradually towards the processor that owns the top four rows of the matrix (i.e., processor #0) depict the data and control flow of the program.
- These patterns are difficult to understand unless a user is knowledgeable of the program and familiar with the patterns that are typically presented in a specific display of a performance visualization tool, such as a space-time diagram.
- Useful displays in conventional visualization tools (such as space-time diagrams) are usually stretched to the limit of their scalability even with moderate system size (16 processors in this case) and immense communication activity involved.

To simplify the process of analyzing the repetitive temporal patterns, we can eliminate the spatial domain and focus on only the temporal domain. Ultimately, the spatial patterns also contribute to an understanding of the behavior of the program. Performance images have been used for this purpose [93, 94, 121, 122, 124]. This type of focusing to support analysis is an important aspect of our multiple-domain analysis approach.

The performance signal from BS is shown with its smoothed version in Figure 6-14. The

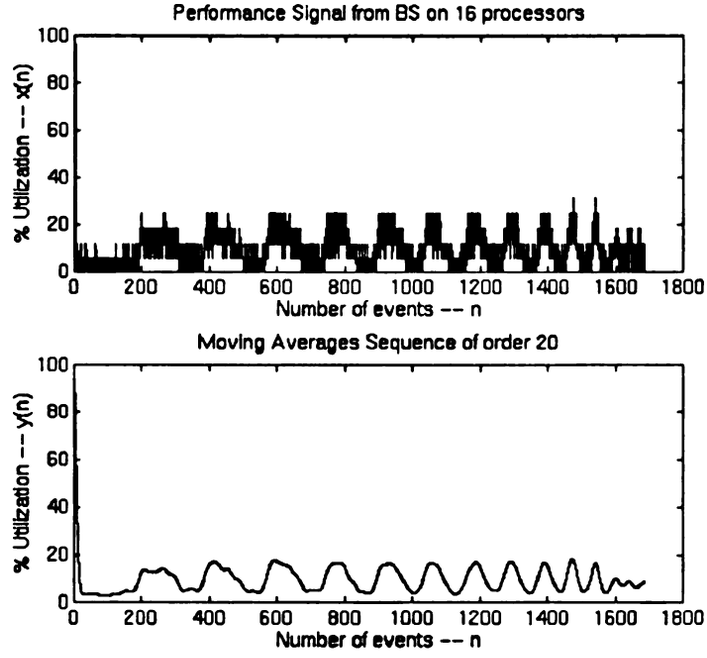


Figure 6-14. Analysis of macroscopic patterns in BS phase signal by using a moving averages filter.

performance signal shows the percent of the system being used throughout the execution

of the program (at the instants of transitions, as this performance signal is obtained from the event domain). Even without processing this performance signal $x(n)$, one can visually appreciate the presence of repetitive patterns, although there is localized "noise" (high variance) obscuring the macroscopic patterns. A moving averages filter of order 20 (arbitrarily selected) was applied to this performance signal and $y(n)$ is the resulting smoothed signal. The smoothing process tries to "impose" stationarity on the performance signal (a time-series) and eliminates its "rapid" variations and "outliers" to emphasize global, average behavior. The following should be noticed in the smoothed signal:

- It shows most of the fifteen "global" (macroscopic) temporal repetitions of the message-passing steps that were indicated in the space-time diagram.
- The macroscopic repetitions are not very clear towards the end, because the number of processors involved in the computation progressively decreases as the algorithm proceeds towards the top rows of the matrix. Therefore, changes in the system utilization (as it is defined) are relatively insignificant and do not show up explicitly in the smoothed signal due to an averaging effect. Similarly, sub-patterns due to the four repetitions at each stage lose their individual identity due to the averaging phenomenon.
- A user does not have to be familiar with the performance signal to see the repetitive patterns shown by an x-y plot of the smoothed performance signal.
- The method scales with system and/or problem sizes due to its rigorous nature.

6.5.3 Estimation of Repetitive Patterns

We begin the analysis of temporally repeating patterns of parallel programs by using performance signals to represent the temporal activity of the system and program. Repetitive patterns are expected in a performance signal due to the presence of loops in the program. Actions within a loop cause variations in certain performance metrics (depending on the nature of the actions). These repetitive patterns can be observed as similar segments of the signal repeating over time, in response to each iteration of a loop. This periodic behavior can be observed using any conventional time-series plot available in a performance visualization tool (e.g., a space-time diagram). As discussed in Chapter 5, this periodicity is not automatically obvious in all cases. Therefore, we use performance signals to analyze these repetitive patterns by applying statistical DSP techniques. We have divided the analysis of repetitive patterns into two parts. The first part is concerned

with the representation, identification and analysis of macroscopic patterns in the whole signal representing the execution of one or more phases of the instrumented program. The second part is concerned with the precise estimation of the repetitive behavior of the program. The combination of these two steps can be applied hierarchically in most program performance analysis situations.

Repetitive patterns are not always easily identified by the removal of variance using averaging. One disadvantage of moving averages that was discussed in the previous section is the loss of precise localized details of the performance signal. If the analysis of patterns is to be hierarchical [29], then a user should be able to locate the macroscopic and average repetitive patterns by using moving averages and then should be able to selectively analyze parts of the program activity more meticulously. If we are able to look at a small segment of the performance signal (this method is termed *windowing* [84] in signal processing), there is still no guarantee that the repetitive patterns will not be hidden by large variance. Therefore, we need to further process this window of the performance signal to locate repetitive patterns (or periodicities). A classical method that is used in statistical signal processing for this purpose is the estimate of *autocorrelation* of the signal [12, 59, 62, 81, 82, 84, 126]. We will explain this method and then provide a couple of examples to establish the validity of the method for the analysis of repetitive temporal patterns of a program.

6.5.3.1 Method: *Autocorrelation*

A widely used method to determine the repetitive (periodic) behavior of a stochastic signal (time-series) is by using an estimate of autocorrelation of that signal [84]. The exact (statistical) autocorrelation is not trivially determined for any stochastic system, therefore, temporal observations are used to estimate the autocorrelation. The unbiased estimate of autocorrelation sequence of the performance signal was defined in Chapter 5 by (5.9) and will be used in subsequent examples to identify the possible iterations of loops in the instrumented programs.

In order to validate the method, we instrumented the loops in two programs discussed in the examples that follow with special markers (called *tracemark* events [45]) so that we could temporally locate the start of an iteration at each processor of the distributed system (nCUBE-2). Whenever a processor enters into the loop, the instrumentation system records a marker with the time-stamp and the number of the iteration (loop index) being performed by that processor, indicating the start of that particular iteration at that processor. We will define and use an *indicator sequence* $\{i(n)\}$ for temporally representing the trace-mark events. In order to obtain the indicator sequence, we look for the trace-mark records in the trace file. Whenever one is found for a processor, at a particular time-stamp n, we extract the information about the loop index represented by that record.

Definition (Indicator Sequence)

The indicator sequence is defined for time-stamps n of a traced programs as:

$$i(n) = \begin{cases} k \text{ , for values of } n \text{ having tracemark events} \\ 0 \text{ , Otherwise} \end{cases} \tag{6.1}$$

where $k \in \{1,2,\ldots, K\}$ is the loop index obtained from the trace-mark record and K is the maximum possible loop index executed by any of the processors.

This definition considers the possibility of generating more than one trace-mark record for a particular time-stamp and a particular iteration (i.e., duplicate markers on different processors). In such a case, only one indicator will be stored. Due to the distributed nature of the program (and especially if there are data-dependent iterations of the loop), it is possible that some processors finish an iteration before the others. Given the definition of indicator sequence, we thus get several points where the amplitude of $i(n)$ is the same for several values of n. This phenomenon will be called "blurring effect" (note the visual appearance of its plots in the following examples, e.g., see Figure 6-17) and will indicate that different processors started the same iteration of the loop at different times. This lack of synchronization due to the distributed nature of the system and SPMD paradigm used should be accounted for while analyzing the estimate of autocorrelation of a performance signal to determine the temporal spacing between two consecutive iterations of a loop.

Indicator sequences were extracted from the trace data of the programs discussed in the following examples to validate the estimated results.

6.5.3.2 Example: *Pi Program*

This example consists of a simple non-nested loop and is presented here to merely serve as a means for illustrating the technique. A second example will be given from a real application program consisting of more complex nested structure of loops. Therefore, the techniques developed here will serve for understanding those cases.

The value of *pi* (3.14.....) can be approximated by evaluating the integrand $4.0/(1+x^2)$ numerically, between the limits of $x = 0$ to $x = 1$. This problem was solved on sixteen nCUBE-2 processors, with each processor working on a small part of the limit. After a local summation is complete, all processors perform an exchange-add process, so that all end up with the final value of *pi*. The exchange-add steps proceed in logarithmic fashion, therefore, there are four exchange-add steps for a system consisting of sixteen processors. The program executes a loop in order to implement these exchange-add steps. The instrumented code (in C) is given in Figure 6-15.

```
tracestart(); /* To set up the system for tracing and open communication
                 channels */
tracenode(500000,0,1); /* To start tracing at all the nodes */
i = 1;
for (j=0; j<lnproc; j++) /* Exchange-add step */
{
     tracemark(j+1); /* Records a tracemark for the node with # of iteration i.e.,
                        j+1 */
     net = iproc ^ i; /* Exclusive-OR: Complement j-th bit. */
     nwrite(&sum, len, net, msgtype, 0); /* Exchange over j-th channel */
     nread(&temp, len, &net, &msgtype, 0);
     sum = sum + temp; /* Accumulate the sum */
     i = i + i;
}
 traceexit(); /* To terminate the tracing */
```

Figure 6-15. Instrumented code for the exchange-add part of the pi program.

The behavior of this program can be represented by a space-time diagram with all its spatial and temporal details as the system size and message-passing involved are not immense. Figure 6-16 shows the space-time diagram with the trace data at the start of the

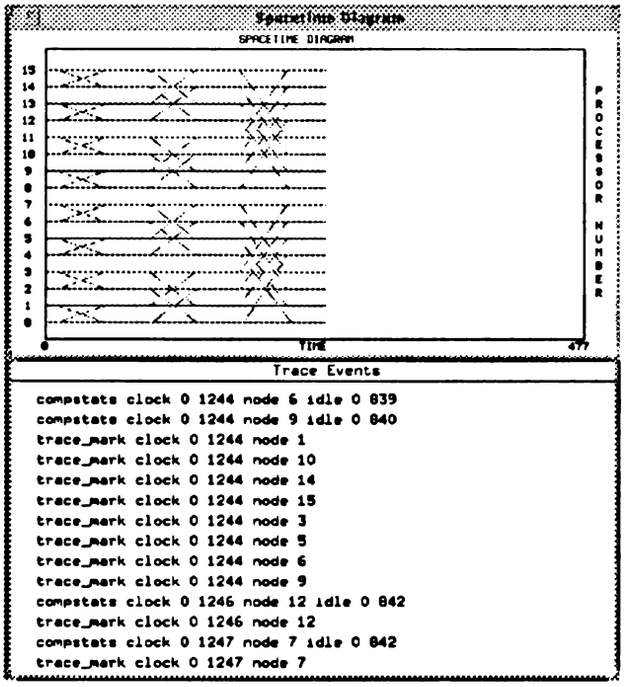


Figure 6-16. Space-time dynamics of the pi program with trace-marks showing start of the fourth iteration.

final (fourth) iteration of the exchange-add loop. The trace-mark records show the time-stamps on processors at the beginning of the fourth iteration. Loop index values are included in the actual trace file as a field of this event record. We will use this information to examine the results obtained from the estimate of autocorrelation of the performance signal corresponding to this computation.

The temporal domain performance signal from the *pi* computation is shown in Figure 6-17 with the estimate of its autocorrelation and the indicator sequence. All three plots are drawn with the same time-scale so that visual comparison is possible. The peaks in the estimate of autocorrelation are referenced by the arrows, which are numbered according to the loop indices identified in the indicator sequence. Although the repetitive patterns are clear in this example, several things can be observed and will be useful in the more practical example given in the next section. These observations can be made:

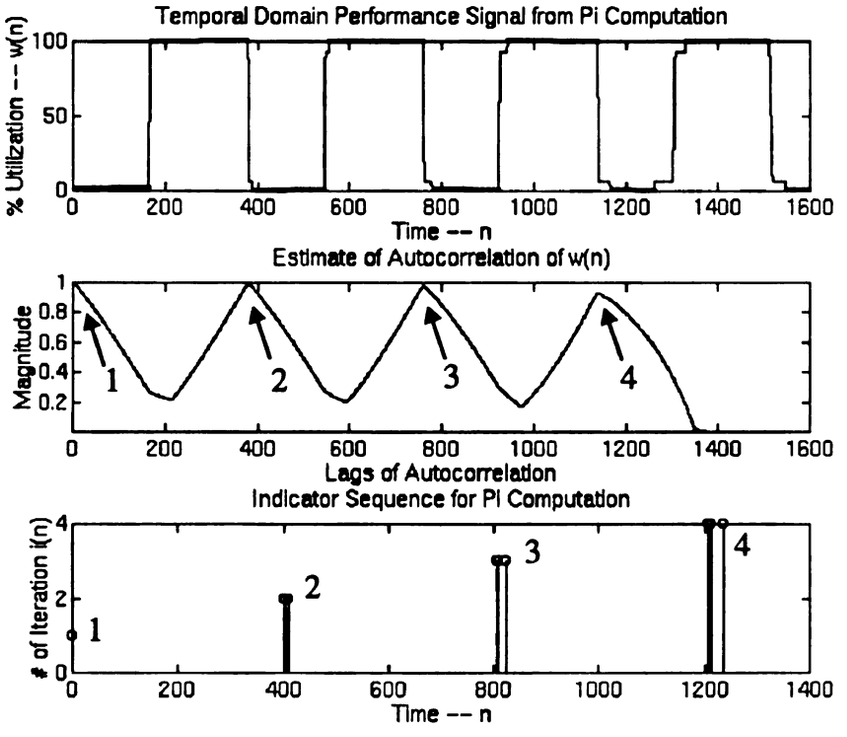


Figure 6-17. Comparison between estimated and actual loop iteration times for pi program.

- The performance signal shows four (identical) repetitions of the system utilization over the entire execution time. The signal starts with zero utilization because all the processors are in a message-passing mode at the same time (due to synchronous nature of the program). Then all the processors complete the message-passing calls and start local computation, which is shown as full (100%) utilization in the performance signal. This pattern is repeated four times, as the loop shown in the code of the *pi* program is executed four times.
- The estimate of the autocorrelation of the performance signal shows four peaks. Normally, if the stochastic process generates discrete data (observations) that are independent of one another, the magnitude of the autocorrelation should be monotonically decreasing. When there is a strong interdependency between two segments of data (i.e., they are almost similar or repeated), then instead of a decrease in the amplitude of correlation coefficients, there will be an increase at the lag where repetition is found. The amplitude of the peak will be maximum (i.e., 1 in this normalized case) if the signal is periodic, i.e., exact repetition of the same sequence is registered. This is the case in this example as there are three instants where the magnitude of the peaks becomes exactly equal to 1, while the fourth shows somewhat lesser magnitude due to the difference of shape of the last part of the performance signal. This shape of the autocorrelation sequence should emphasize two points: (1) there were four repetitions of a loop that contained instructions that were executed in an identical fashion on each processor (i.e., not dependent on any local information); and (2) the time between any two iterations (the period) is equal to the distance between their corresponding peaks in the autocorre-

lation. The period is approximately 400 clock cycles (i.e., samples) as shown by the autocorrelation sequence. In this case, the periods are almost identical due to the synchronous nature of the computation.

- The indicator sequence confirms the information given above. The first bar at $n = 1$ has an amplitude equal to 1, therefore, it shows that the first iteration started at all the processors simultaneously at the first clock cycle (because it is not "blurred"). The second iteration starts at $n \approx 400$ as the amplitude of the bars is equal to 2. The blurring effect shows that some processors started the second iteration earlier than the others. The period between the first and second iterations was ≈400 clock cycles. The third iteration has been shown to start at $n \approx 800$, while the fourth iteration starts at $n \approx 1200$. The blurring effect increases with time, as the effect of initial brute-force synchronization of the system keeps on decreasing due to the distributed nature of the computation.

TABLE 6-2. Summary of results of estimating loop timings for the *pi* program.

Loop Index	Estimated Time of Iteration (clock cycles)	Actual Time of Iteration (clock cycles)	% Error Range
1	1	1	0
2	382	401 – 411	4.74 – 7.06
3	761	804 – 824	5.35 – 7.65
4	1140	1207 – 1237	5.55 – 7.84

The data given in Table 6-2 (based on Figure 6-17) indicates that the estimated information (from the autocorrelation sequence) compares favorably with the actual information provided by the indicator sequence (obtained by brute-force processing of the trace data, in this case) as there is negligible error. This example illustrates that it is constructive to apply an estimate of autocorrelation to identify repetitive behavior and associate it with the loops in the program. In the next example, we will apply the technique developed here to a case where repetitive patterns are not so obvious.

6.5.3.3 Example: *Linear Solver Program*

The example discussed here involves the GE phase of the same solver program used in section 6.5.2.2. Recall, the data is distributed so that each processor owns four contiguous rows of the coefficient matrix (i.e., (BLOCK,*) data distribution). The GE algorithm proceeds from the top row of the matrix to the bottom row and results in an upper triangular matrix [4, 37]. At each step, the processor that owns the row being used for the current iteration broadcasts the location of the pivot element to all other processors. Then,

the pivot row is passed to all other processors, followed by the element of the constant vector corresponding to the pivot row. After each step, the "active" rows of the matrix reduce by one. Again, each repetition of these steps was recorded with a trace-mark event record which will be used for the analysis presented in this section. The space-time dynamics of this computation are represented in Figure 6-18. There are a number of patterns shown by this display as in the case of the BS phase analyzed earlier. We make

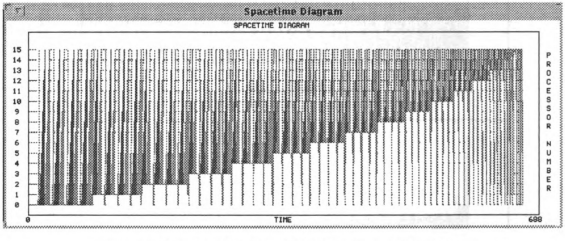


Figure 6-18. Space-time dynamics of GE phase of Linear Solver.

the following comments about this space-time diagram (some of them similar to those given for the space-time diagram shown in Figure 6-13):

- The patterns are repetitive, with the dominant macroscopic pattern repeating fifteen times. Each step in this pattern contains four similar repetitions, due to each processor owning four rows of the matrix. In each of these four "localized" (microscopic) patterns, there are three sub-patterns. The first sub-pattern shows a broadcast from the processor that owns the pivot row at a step. The second sub-pattern shows the message-passing as the elements of the pivot row are sent to all the processors that own lower rows of the matrix. The third sub-pattern shows the message-passing as an element of the constant vector is passed to all the processors owning lower rows.
- Similar patterns shifting gradually towards the processor that owns the bottom four rows of the matrix (i.e., processor # 15) depict the data and control flow of the program.
- Again, it should be noted that these patterns are difficult to understand unless a user is knowledgeable of the program and familiar with the patterns to be expected.
- Once again, the space-time diagram has been stretched to its limit of scalability, and patterns are too "dense" so that they are difficult to comprehend. If the time-scale is expanded to counter this problem, then the overall macroscopic representation and realization of the patterns discussed above is lost.

Figure 6-19 shows the (event domain) performance signal for the GE phase with the estimate of its autocorrelation. In this case, it is clear that the performance signal does not

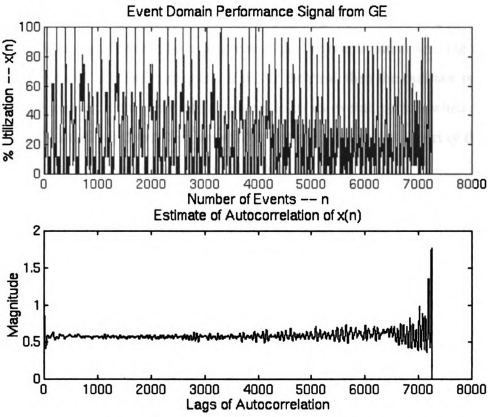


Figure 6-19. Estimate of autocorrelation of the whole GE performance signal.

show any patterns that are readily observable. Therefore, the use of autocorrelation is justified. The estimate of autocorrelation shows several peaks indicating repetitive patterns in the performance signal that are not obvious directly. However, the phenomenon of increasing variance with larger lags using the unbiased estimate of autocorrelation can be observed in the estimate of the autocorrelation of the signal. Therefore, the correlation coefficients at higher lags can not contribute towards our understanding of the repetitive (looping) behavior in this program. Moreover, the number of samples in the whole event domain performance signal is very large, and dependencies (and hence the precise estimate of locations of repetitions) can not be appreciated individually from this information. Therefore, we need to look at a small portion of the signal (known as a

"window" in DSP [84]) and determine the estimate of its autocorrelation. Usually, the length of this window depends on the capabilities of the available processing resources. For our purposes, the window size should be kept small enough that the variance of the estimate of autocorrelation does not increase by a large amount at large lags and the amplitude at those lags does not become greater than the amplitude at zero lag (which has been normalized to 1 in these examples). If this is the case, then the estimate is consistent and reliable. However, if the performance signal loses its overall pattern when subdivided into sufficiently small windows, then we will be forced to use only that part of the estimate of autocorrelation where amplitude does not exceed 1.

In order to be more precise, we divide the whole GE signal into windows, each of length equal to one tenth the length of the whole signal (appropriately chosen for this signal). The estimate of the autocorrelation of the first window of the signal is shown in Figure 6-20 with the corresponding portion of the indicator sequence. Although the performance signal shown in this figure is not strictly periodic due to the asynchronous nature of the program, the time between any two repetitions is approximately same, as seen both in the estimate of autocorrelation and in the indicator sequence (or even the actual performance signal). The indicator sequence shows (by blurring effect) that the iterations after the first iteration are not synchronized. This causes variations in processor utilization that do not fit an exact pattern, and thus the peaks representing repetitions are not sharp. For instance, the temporal behavior associated with the second peak is difficult to determine because it does not show a sharp, unique maximum. However, by comparing the time-scales of the estimate of autocorrelation with that of the indicator sequence, we see that the range of the temporal locale of an iteration as shown by each of them is almost equal. Therefore, both the estimated information (by autocorrelation) and the actual information (from the indicator sequence) do not identify the repetitive behavior of the program with absolute precision, in this case. This comparison is given in Table 6-3 for the four iterations shown in the window of Figure 6-20. The error is shown to vary from a negative to a positive value to reflect the position of a peak relative to the corresponding indicator sequence range.

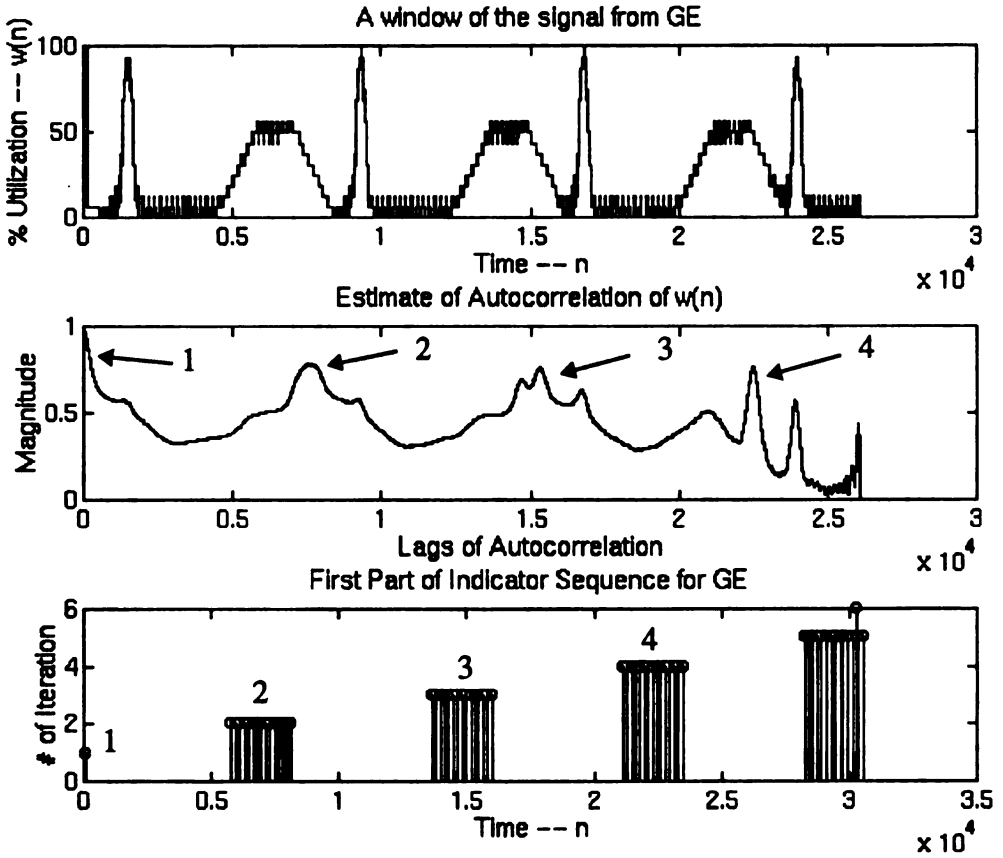


Figure 6-20. Estimate of autocorrelation of a window of GE performance signal.

TABLE 6-3. Summary of results from estimating loop timings of GE phase.

Loop Index	Estimated Time of Iteration (clock cycles)	Actual Time of Iteration (clock cycles)	% Error Range
1	1	1	0
2	7,694	5,780 – 8,125	-33.11 – 5.30
3	15,333	13,632 – 15,982	-12.47 – 4.06
4	22,504	21,119 – 23,469	-6.56 – 4.11

In the context of this example, we have taken a closer look at the blurring effect. Specifically, the blurring effect was removed by placing a barrier synchronization at the end of each loop. Therefore, every processor waits until all the processors have finished the iteration of the loop, and then all of them start the next iteration. The same window of the performance signal obtained from this computation was analyzed with its estimate of

autocorrelation and the indicator sequence. The results are shown in Figure 6-21. The first obvious difference in this figure is the removal of the blurring from the indicator sequence as expected, because all the processors were iterating the loop synchronously. The

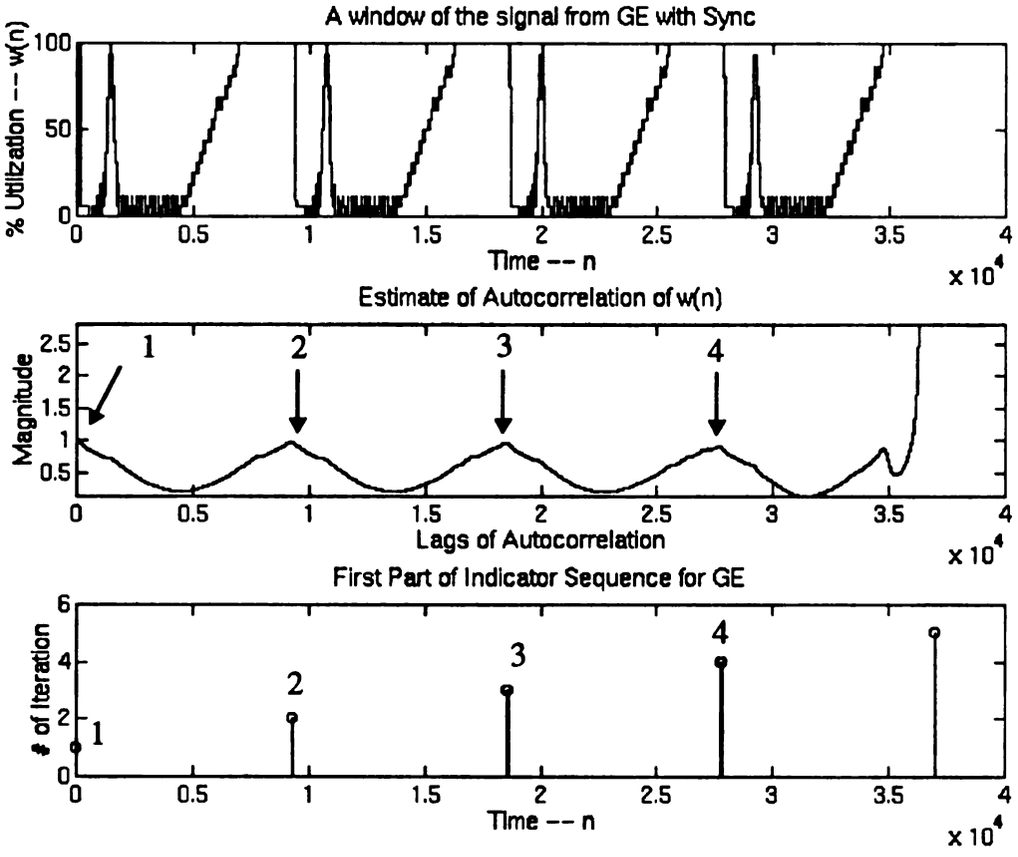


Figure 6-21. Estimate of autocorrelation of a window of GE performance signal with synchronization.

occurrence of the low-level synchronization call [45] is realized in the performance signal by its effect of introducing segments having 100% system utilization (essentially by definition of the instrumentation). As expected, now the peaks are very sharp and aligned with the iteration times shown by the indicator sequence. The results given in Table 6-4 can be compared with Table 6-3—there is now minimal error in the estimation. The period of all the repetitions is approximately 9,000 clock cycles, which can be observed from any one of the three plots shown in Figure 6-21.

TABLE 6-4. Summary of results from estimating loop timings for GE phase with synchronization.

Loop Index	Estimated Time of Iteration (clock cycles)	Actual Time of Iteration (clock cycles)	% Error Range
1	1	1	0
2	9,248	9,281 – 9,320	0.36 – 0.77
3	18,500	18,540 – 18,583	0.22 – 0.45
4	27,740	27,774 – 27,816	0.12 – 0.27

6.5.4 Estimation of Template Patterns

We can enhance the efficacy of the pattern estimation technique that was used in the previous section by using our knowledge of various patterns resulting due to commonly used high-level message-passing calls. This knowledge can be incorporated into the analysis of patterns by using *template signals* for commonly used patterns, such as shown in the examples of section 6.5.1. These template signals can be compared with the temporal or event domain signals corresponding to the execution of a program to estimate the possibilities of finding the occurrence(s) of the same pattern. If the user is knowledgeable about the nature of high-level message-passing calls used in the program, then ostentation process will result in locating the temporal location of the occurrence of these patterns, that is useful for debugging and analyzing the behavior of the program. A template signal is defined in the following.

Definition (Template Signal)

It is a performance signal representing a high-level message-passing activity. The template signal can be obtained either from event domain or temporal domain.

Template signals will be selected from well-known and often-used high-level communication routines that can be found in various programs. It should be noted that the shape and characteristics of template signals corresponding a high-level message-passing call obtained from different parallel systems (e.g., using different system sizes) might differ from one another. We will use estimate of cross-correlation between performance signal and the template signal for the purpose of matching a performance signal with a template to determine the degree of similarity between the two at all event domain

locations. This process will result in confirmation or contradiction of user's expectations to find certain patterns (due to specific high-level message passing calls) during the execution of the program.

6.5.4.1 Method: *Cross-correlation*

The estimate of cross-correlation sequence is a generalized form of the autocorrelation sequence. The cross-correlation of two performance signals is defined by the equation (5.10) in Chapter 5. We had mentioned two uses of the estimate of cross-correlation: to estimated the likelihood of occurrence of a known high-level communication patterns and to compare two different programs. Use of both of these techniques is shown in the following by examples from the linear solver.

6.5.4.2 Example: *Linear Solver*

The GE phase of a solver of a linear system of equations was discussed as an example of using estimate of autocorrelation to identify temporal location of each iteration. It was discussed that each of the iterations uses a broadcast call to pass the location of pivot elements to all other processors. Broadcast is a commonly used high-level message-passing function that is often used in various types of application programs. Due to the general nature of this communication function, it might become important for debugging a program that uses this call. If the template signal for broadcast for a particular parallel system (using a specific number of processors) is known, it can be correlated with the performance signal to find out whether the broadcasts were present at desired temporal locations or not. The performance signal from GE phase of the linear solver was correlated with the template signal for broadcast on 16 processors, shown in Figure 6-22. The number of peaks in the estimate of cross-correlation of the two indicate the strong likelihood of occurrence of broadcast calls in the GE phase at the points where magnitude was maximum. A magnitude of one indicates a certain occurrence of the same pattern at that lag (from which actual time-stamp can be found out) and negative magnitude indicates strong dissimilarities between the two signals at that lag. Since the dimension of

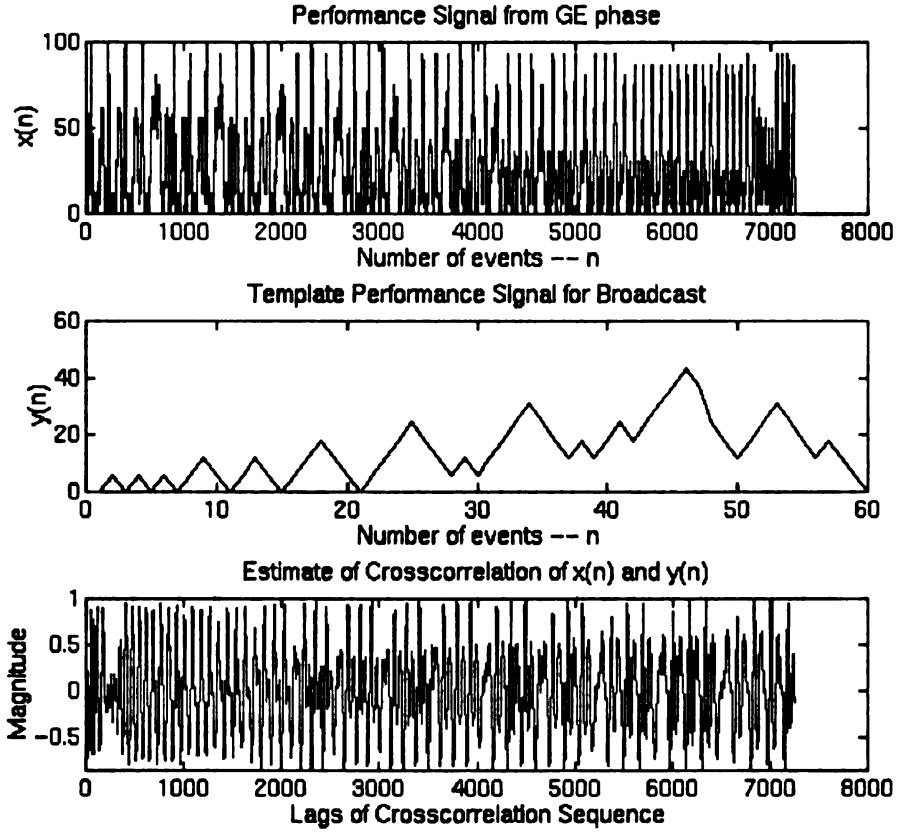


Figure 6-22. Estimation of occurrence of broadcast calls during GE via estimate of cross-correlation.

the matrix was 64, there were 64 broadcasts in this phase, which are indicated by the estimate of cross-correlation sequence.

The precise estimate of the presence of the broadcast can be accomplished by selecting a small window of the performance signal and estimating its cross-correlation (actually cross-covariance) with the template signal from the broadcast among same number of processors. This estimate can again be validated against the exact information in the form of an indicator sequence that indicates a call to the broadcast function. The indicator sequence is defined in the same way as given by equation (6.1) except for the fact that it indicates a particular occurrence of the broadcast call on a particular processor. Definition of this indicator sequence follows.

Definition (Indicator Sequence for Broadcasts)

The indicator sequence is defined for time-stamps n of a traced programs as:

$$i(n) = \begin{cases} k, & \text{for values of } n \text{ having tracemark events} \\ 0, & \text{Otherwise} \end{cases} \quad (6.2)$$

where $k \in \{1,2,..., K\}$ is the number of broadcast call executed on a processor obtained from the trace-mark record and K is the maximum possible number of such calls executed by any of the processors.

Once again, this definition also considers the possibility of generating more than one points where the amplitude of $i(n)$ is the same for several values of n. This blurring effect will indicate that different processors executed the same broadcast call at different global times. Unlike the case of estimation of repetitive patterns, this lack of synchronization will not affect the estimate of cross-correlation between template signal and the program performance signal, as this phenomenon manifests in the template signal as well due to asynchronous nature of computation.

The GE phase of the linear solver was modified to include trace-mark calls every time a processor called broadcast function. A window of the temporal domain performance signal from the GE phase was selected as was compared with the broadcast template (also a temporal domain performance signal). We have selected temporal domain signals to estimate their cross-correlation, so that, we could compare the estimated times of occurrences of broadcasts with the actual times from the indicator sequence. The results are shown in figure 6.23. The estimate of cross-correlation identifies four occurrences of the broadcast calls in the selected window of the performance signal by four peaks. These instances are indicated by arrows and numbers in estimated signal and the indicator sequence. A "trough" immediately after the peak in the estimate of cross-correlation points towards the fact that the template signal decreases towards zero towards the end (after the broadcast), whereas the actual signal increases towards higher values after the end of a particular broadcast. This is due to the conditions under which template signal was obtained. This particular template signals was obtained by tracing a single broadcast initiated by processor number 0 to the other 15 processors. After tracing this broadcast, the

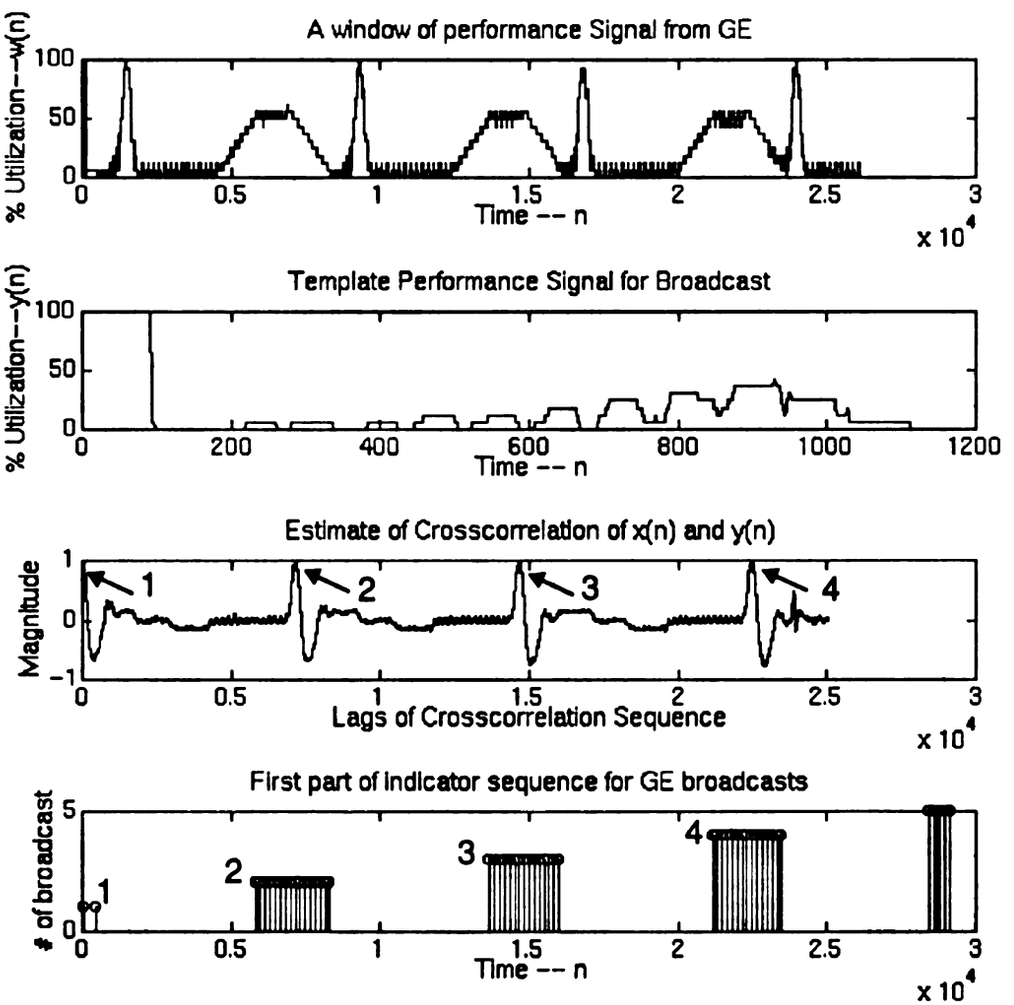


Figure 6-23. Precise identification of broadcast from GE with "proper" alignment of time

tracing stopped on all the processors, so they were no longer utilized. Hence, the template signal approaches zero towards the end. In the actual signal from the GE phase, the program continues after the broadcast. Thus, the utilization increases towards higher values after the broadcast. That is, at the end, the template matches negatively with the actual signal, a fact indicated by a value closer to -1 in the estimated cross-correlation signal.

The numerical values are compared in Table 6-5. As it can be appreciated visually, the estimate is quite close to the actual timings shown by the indicator sequence. The error seems to be very large for the first broadcast but it is only due to the manner in which error

range is calculated. We compare the difference between actual and estimated value against the actual value, therefore, with small actual value at the start compared to a larger estimated value shows up as a very large error. Otherwise, the error seems to be reasonable.

TABLE 6-5. Summary of results from estimating broadcast timings for GE phase.

Number of Broadcast Call	Estimated Time of Broadcast Call (clock cycles)	Actual Time of Broadcast Call (clock cycles)	% Error Range
1	24	49 – 448	51.02 – 94.64
2	7,165	5,830 – 8,282	-22.89 – 13.486
3	14,668	13,675 – 16,009	-7.26 – 8.37
4	22,457	21,165 – 23,485	-6.10 – 4.37

The cross-correlation can also be used to identify the possibility of a "suspected" pattern in a performance signal from a program. If the cross-correlation shows peaks with large positive magnitudes, then there is a strong likelihood for the pattern to be present in the signal. This procedure can be particularly helpful to identify the patterns that ensure good performance (or bad performance) or can be used for debugging to identify a "failed" (or a "correct") patter. For instance, suppose it is suspected that BS phase of the linear solver uses a broadcast to pass an element of the solution vector to the other processors (it actually does not uses message-passing for this purpose). The objective is to find out whether this pattern is there in the BS performance signal or not. The cross-correlation of the broadcast template and BS performance signal is shown in Figure 6-24. The dominant negative peaks and small magnitude of the positive peaks shows that there is very small likelihood of finding a broadcast pattern in this phase of the program. The result agrees with the actual situation.

6.5.4.3 Example: *Comparison between Data Distributions*

The linear solver that we have discussed in several examples can be implemented by using different coefficient matrix data distribution techniques. The technique that was used in the examples so far, is called (BLOCK,*) [36] which involves the distribution of a block

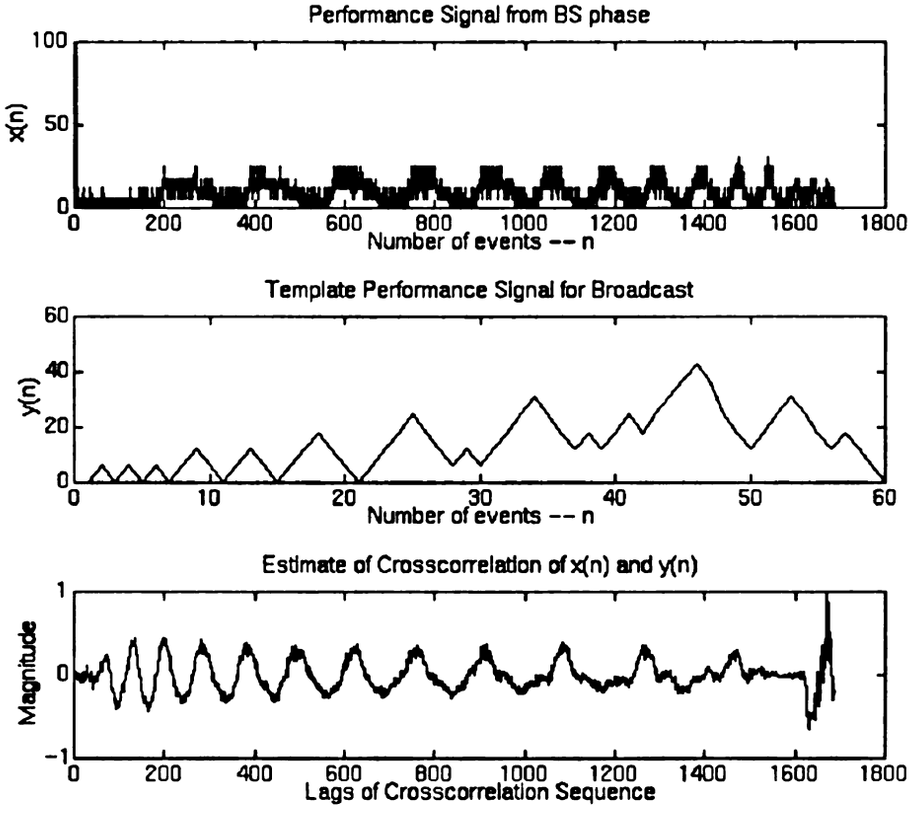


Figure 6-24. Estimation of the possibility of broadcast in BS phase of linear solver.

of rows of the matrix to each processor. There are other data distribution techniques that can be used for this purpose, and perform better than the above technique. One such technique involves dividing the matrix into blocks of rows that are a constant distant apart from each other. This technique is called (CYCLIC,*) [36]. If we have the performance signal from two implementation of the GE phase, one using (CYCLIC,*) and other using (BLOCK,*) data distribution, then the estimate of cross-correlation between the two signals can indicate the points where the two signals have similar patterns and where they differ the most. Figure 6-25 shows the two signals and their correlation. Both implementations used 16 processors and the dimension of the linear system that was solved in both cases was identical, having a dimension of 64. Therefor, there were similarities in the two signals, indicated by positive maximums in the estimate of cross-correlation. The differences were mainly due to different data access patterns due to the

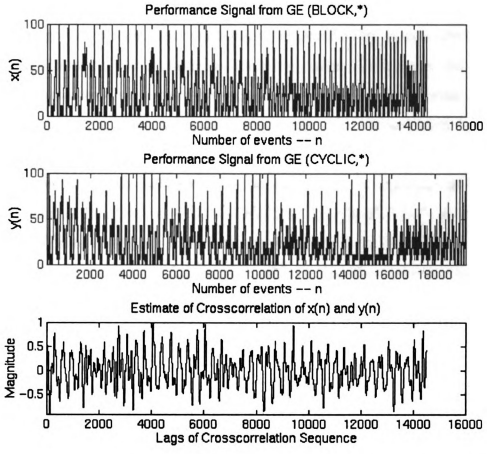


Figure 6-25. Comparison of GE phase implemented by using two different data distributions.

distributions, therefore, the differences were also expected. The precise location of the similarities and differences can be used to locate the differences in the two programs, if a hierarchical analysis is required. This can be accomplished by using small windows of the signals, at each step of comparison.

6.5.5 Identification of Arbitrary Patterns

Characterizing the behavior of the expected patterns in a program using template signals might not always be possible. However, user still might have some notion about the patterns. Estimation of cross-correlation between a known pattern and the actual pattern depends on the availability of the user's knowledge of the patterns as a template signal. When template signal is not available, patterns can be characterized by other information.

There are several patterns of the programs that are best specified by the amplitude of the performance signals (i.e., the magnitude of a performance metric). A program can cause the system to go to a certain state, which might be identified by an excessively large (or small) value of a metric that characterizes the system states. For instance, the events that change the system state such that the system utilization becomes maximum (100%) or minimum (zero) might show the extreme conditions of communication load on the network, or load balance (or imbalance) among the processors. The amplitude need not be an outlier to provide useful information to the user—it can be any value that has some special significance in a given case. This is valuable information about a program which should be applicable for identification of patterns even when they can not be characterized by a template signal. In such cases, a threshold operator can enhance the user's capabilities to explore the arbitrarily defined patterns that depend on the amplitude of the performance signals.

6.5.5.1 Method: *Thresholding*

We have defined a threshold operator (or filter) on the performance signals $x(n)$, so that we could identify the instances of a specific events that are coupled with a particular amplitude of the performance signal. Thresholding was defined in Chapter 5 by equation (5.11). The use of threshold filter is useful for two purposes:

1. Identification of synchronization points in a program that have a specific pattern of making the system utilization maximum at the point of synchronization. These points can be isolated from others to find out whether all the synchronizations were being implemented correctly or not.
2. Identification of the points of minimum utilization is possible by setting the value of threshold equal to the minimum value of the system utilization metric being used (or any other appropriate metric with appropriate threshold). This procedure results in temporal localization of the performance bottlenecks.

We perform these two analyses in the following examples obtained from a particular implementation of the linear solver using 16 nCUBE-2 processors.

6.5.5.2 Example: *Synchronization Points in GE Phase*

There are several points in GE phase of the linear solver where processors are synchronized after an iteration of the main loop of this phase. Some of these synchronizations are explicitly there, while others are only implicit. The explicit synchronization is implemented at the start of the tracing to synchronize the clocks of all the processors. The rest of the synchronizations are implicitly implemented when all the processors are able to finish an iteration of the main loop and start local computations.

Figure 6-26 shows the performance signal from GE Phase with the thresholded signal (according to the above definition) that identifies all the synchronization points. The first occurrence is related with the explicit synchronization call implemented by instrumentation system (PICL). It can be noticed that the next few instances of synchronization appear at irregular intervals, which means that some iterations of the loop ended with implicit synchronization of all the processors while others could not be synchronized due to SPMD programming paradigm being used. The thresholded signal also points out an interesting phenomenon by showing all the occurrences of synchronizations during the initial part of the program. This means that the synchronous nature of the message-passing shows up for some time but then diminishes due to: variable message-passing times; variable computation dependence on the local data of a processor; and variable number of processors having "active" elements of the matrix. Therefore, the (implicit) synchronization is lost after some iterations, when the number of processors having "active" elements of the matrix reduces significantly.

6.5.5.3 Example: *Identification of Arbitrary Patterns*

An arbitrarily set threshold can identify user specified information within the performance signal from a program. For instance, if we want to find out the instances where the system utilization decreases to its minimum value (i.e., none of the processors are busy computing), it can be identified by using the threshold filter defined above with a threshold value of zero. The result of implementing this filter on the performance signal from GE phase is shown in Figure 6-27. It can be appreciated that there is a very large number of

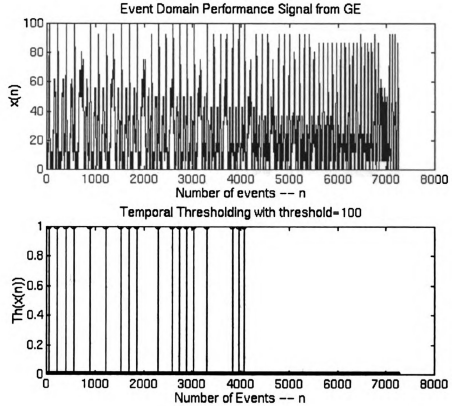


Figure 6-26. Identification of instances of synchronization in GE phase of the linear solver.

instances where system utilization hit its minimum value. This is expected due to the amount of message-passing involved for this phase of the linear solver. Any other user defined values, such as 25%, 50% etc. of system utilization instances can be located from the performance signal using appropriate value of threshold.

6.6 Analysis of Patterns using Multiple-Domains and Views

Parallel program patterns are not restricted to the temporal domain patterns that are obtained from time-series using the methods explained in previous sections. Temporal domain patterns are concerned with the temporal behavior of the program. There are other aspects of the program behavior that might have specific patterns if represented in other domains or using other views. These patterns can be used to characterize the behavior of a particular program on a particular parallel system. While temporal patterns characterize the dynamic behavior of a program, the patterns that we will discuss in this section

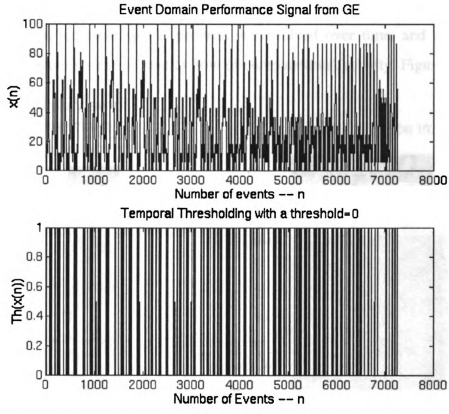


Figure 6-27. Identification of instances of minimum system utilization during GE phase.

characterize (or summarize) the overall behavior in spatial and temporal domains. We will discuss two types of patterns in this section: (1) spatial domain patterns represented by performance images; and (2) patterns showing load-balance of the system during the execution of a program. By some examples, we will show that some of these patterns might be unique and can be used for comparison of overall behavior and performance of two programs, implemented differently.

6.6.1 Analysis of Spatial Domain Patterns

We have used performance images [93, 94, 122] for characterization of the spatial domain. It was shown in [93] that patterns of performance images, representing spatial behavior of the system at a particular time during the execution, can be analyzed using digital image processing techniques, such as image subtraction, enhancement and thresholding.

Cumulative system utilization metric can be used to characterize the spatial domain behavior of a program, which is also dependent on the temporal behavior as well. Therefore, it can represent spatial patterns that evolved over time, and can show the dynamics of system states and spatial flow of computational activity. Figure 6-28 shows

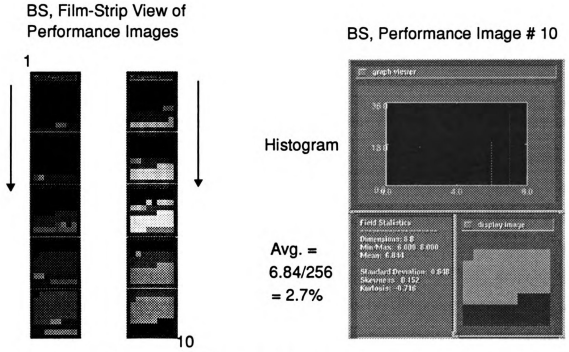


Figure 6-28. Performance images taken after equal intervals during BS phase and analysis of last image.

ten performance images taken during the BS phase using 64 processors of nCUBE-2 whose temporal domain patterns were discussed in section 4.1. Performance image spatially represents a processor in a two-dimensional grid, with the first processor mapped to the top-left corner and the last processor mapped to the bottom-right corner. Figure 6-28 also shows the statistics and histogram of the last performance image. If we look at the "film-strip" of the images, it can be observed that:

- The computation activity (represented by cumulative utilization metric) starts from the processors that own bottom rows and shows up in the form of red color at the bottom-right corner of the performance image.

- The computation activity keeps on moving "upwards" i.e., towards the processors that own upper rows of the matrix, as the computation progresses till the final performance image. Lower computational activity shows up in a shade of blue color while higher computational activity gets mapped to red color in the color table of AVS.
- Same information can be obtained by analyzing the performance image # 10 that was taken at the end of the BS phase. It shows three distinct "clusters" of computational activity. The processors that own bottom rows are utilized less compared to the processors that own top rows and participate in the computation till the last element of the solution vector is calculated from the first row of the matrix. Therefore, it shows the control flow of the computation towards the processors that own top rows from those containing bottom rows.
- The performance image is a scalable representation and is useful to represent the computational activity and the control flow patterns of programs executed on MPP systems.

In cases where performance images represent computation on MPP systems, there are various image processing and enhancement techniques that have been shown to be useful that can be applied to understand the patterns of the program [93, 94, 122]. A useful analysis technique that was shown in [93] is the use of thresholding, which can be used interactively on performance images to locate any spatial patterns of the program. For MPP systems, where instrumenting the whole execution of a program of all the processors is not trivial, the use of performance images coupled with image processing techniques is one of the very few techniques available for the performance analysis.

6.6.2 Frequency Distribution Patterns

The use of event-frequency distribution to characterize the statistical behavior of the program was discussed in Chapter 5. The temporal domain performance signal of a program representing the values of the system utilization at each discrete instant of execution time, can be considered univariate discrete data. Its discrete values range from zero to 100%. If we draw the occurrences at each of the discrete values in this range, it will show the frequency density function of this metric (from which frequency distribution can be calculated). Figure 6-29 shows the distribution of system utilization metric of GE phase of the linear solver using (BLOCK,*) and (CYCLIC,*) data distribution strategies. These curves (histograms) show the load-balance of the system during these programs. The patterns in both cases are almost similar showing the maximum of the density

function at lower system utilization values. This indicates a poor system utilization during both implementations of this phase of the linear solver. The reason for this poor load-balance is the excessive communication involved to implement this phase that deteriorates the system utilization. This pattern is unique for this phase of the linear solver and can be used to either characterize or statistically model this phase by fitting the frequency distribution (that can be obtained from the histograms shown in Figure 6-29) to a theoretical distribution.

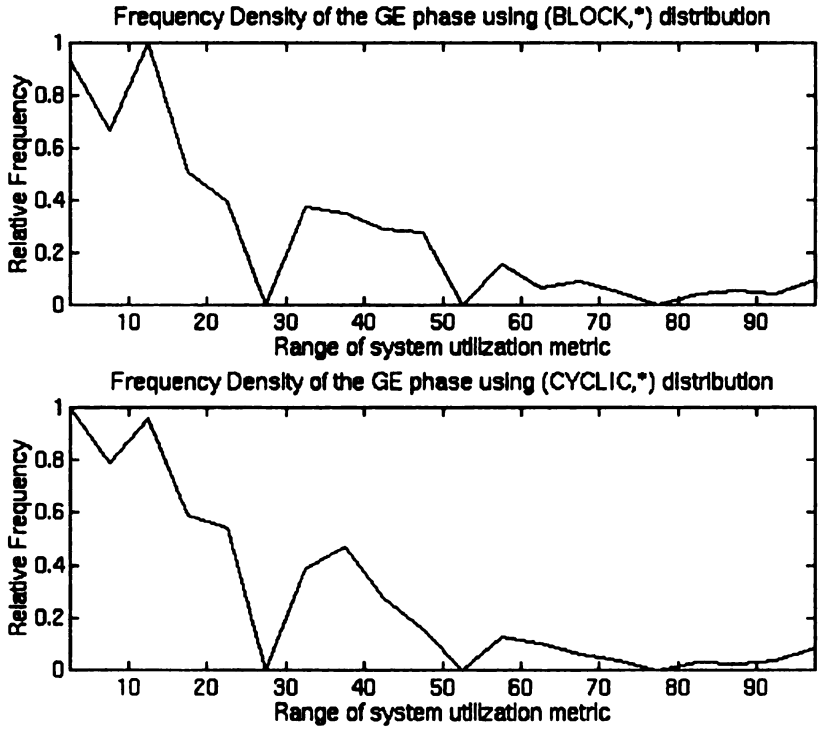


Figure 6-29. Frequency density histograms of two implementations of GE phase of linear solver.

Figure 6-30 shows the frequency density functions of temporal domain performance signal from BS phases of the linear solver, again using two different data distribution strategies. In this case, the histograms representing the density functions are different to some extent. The histogram of BS phase using (BLOCK,*) data distribution shows that all the occurrences were concentrated around low system utilization values, throughout the program. This shows poor system load-balance during this implementation of BS. On the other hand, the BS implemented by using (CYCLIC,*) data distribution shows some

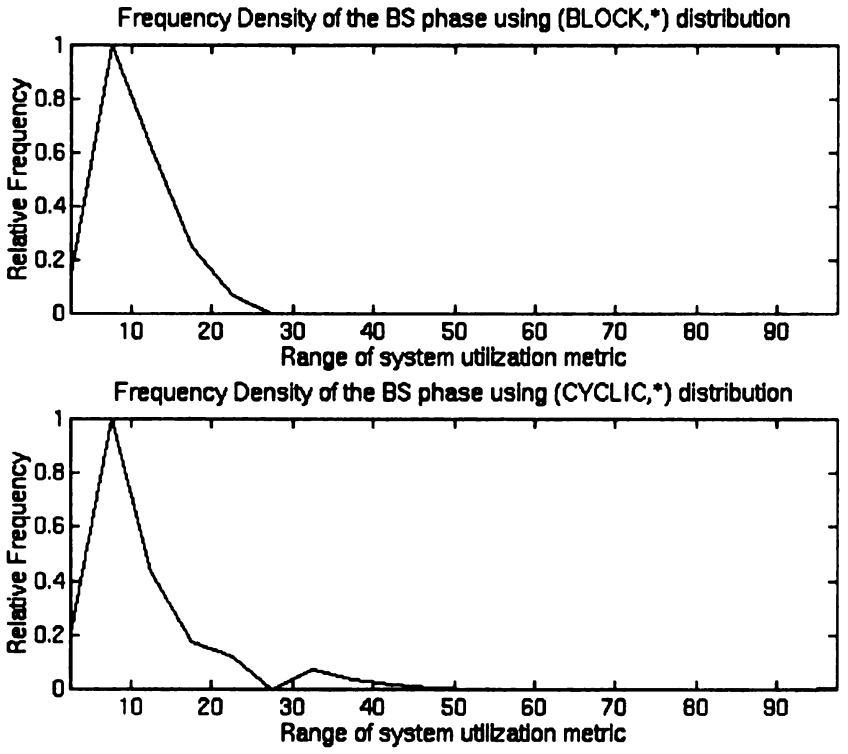


Figure 6-30. Frequency density histogram of two implementations of BS Phase of linear solver.

improvement in load-balance compared to (BLOCK,*) implementation, but overall load-balance is still not much different. Therefore, this pattern not only characterizes the program statistically, but also provides qualitative information about the performance of the program.

The spatial domain representation of the performance data in the form of performance images can also be considered a transformation to the univariate data, i.e., color table indices. The frequency distribution of these color indices can be calculated from their relative frequency shown by the historgram. The patterns in this histogram can identify any spatial clusters (of the processors) with respect to the metric being represented by colors. Figure 6-31 shows the performance image from GE phase of the linear solver with its histogram. In GE phase, it is clear that the values of color indices are spread out all over the range of the color indices. Therefore, this histogram can be considered as showing

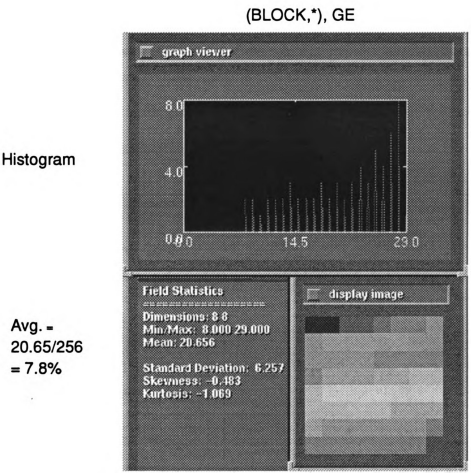


Figure 6-31. Performance image from GE phase with its histogram.

load-balance in spatial domain. The patterns of this histogram can also characterize and statistically model the computation in spatial domain.

6.7 Enhancing Aural Analysis

The multimedia concepts are recent additions to the area of program performance visualization [13, 39]. Program behavior can be mapped to sound (called auralization or sonification), and can be used for program debugging and can complement algorithm animation. Sonification of a program can be considered as a very long sequence of a discrete-time (aural) signal. DSP techniques are potentially useful for analysis and synthesis of auralizations. Work continues for synthesis of aural signals, but we demonstrate the ability to analyze these signals. Auralization refers to the representation

of program events using sound to convey information about program behavior [39]. As an example, consider a send-receive sound mapping which uses different notes to emphasize communication behavior (the details of the mapping and the program it was applied to can be found in [39]). The graph of a sound signal depicting all notes present in the auralization is shown in Figure 6-32. This sound signal is a performance signal and can be

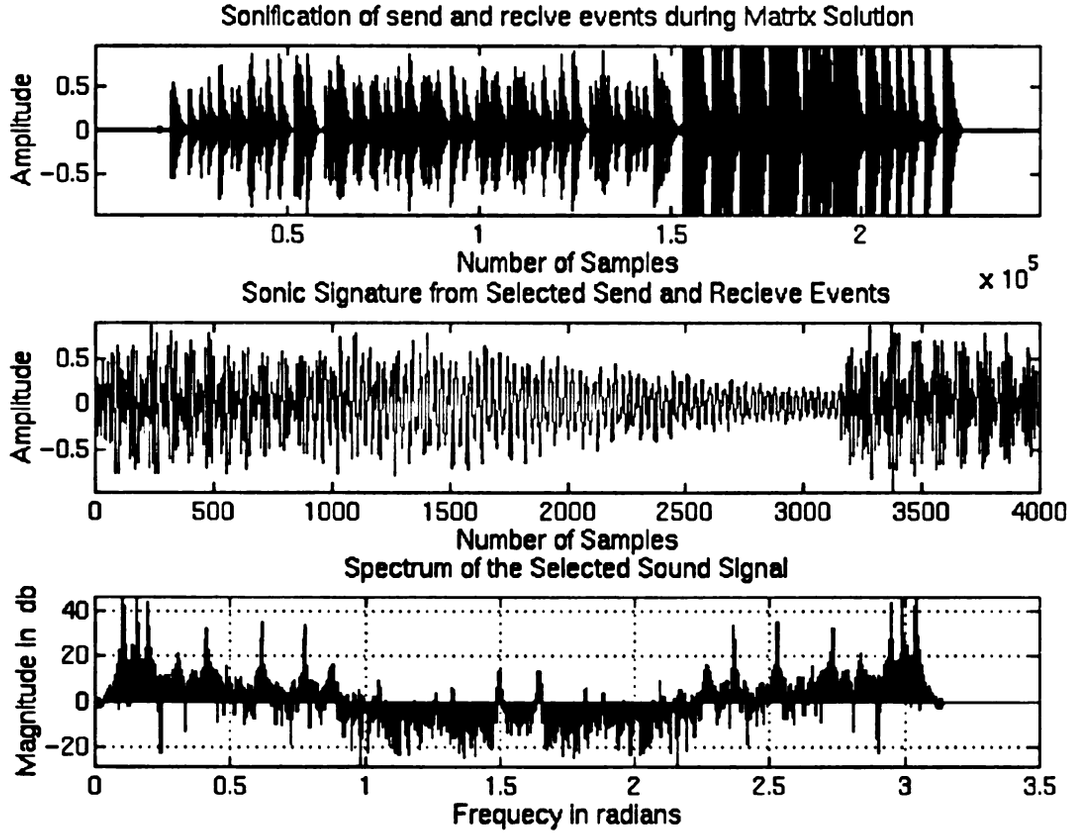


Figure 6-32. Sound signals representing an auralization in the time and frequency domains.

analyzed using DSP techniques. The second graph is a "windowed" portion of the first (a rectangular window was used, although other types are available). Windowing is necessary if a portion of the signal is to be analyzed (e.g., as the remainder is being generated). The third graph is the spectrum of the signal shown in the second graph. It shows the same information as a stereo's graphical equalizer does (except at a higher resolution, i.e., with more frequency bands). Graphical representation of an auralization in

the time domain (i.e., a time-based signal) lets the user visualize the sound patterns that are being heard. For example, the relative amplitudes, durations, periodicities, etc. of the notes are observable and can be analyzed. The spectrum of the signal in the frequency domain shows the dominant frequencies, which reinforces the perception of various pitches (frequencies) heard in the auralization.

The discussion of analysis of behavior and patterns of a program, in various domains presented in this chapter, has used the multiple-domain analysis approach that was presented in Chapter 5. The multiple-domain analysis approach can be used in conjunction with typical, conventional analysis methods and displays in a systematic manner. This will ensure scalability, extensibility and efficacy of the program performance analysis in most of the cases. The purpose of multiple-domain analysis approach is not to prove that the conventional methods and typical displays do not work in all the cases. These methods are effective in their own right. However, the use of multiple-domain methods and analysis of patterns will only supplement the user's knowledge of the program behavior obtained from conventional methods, in cases where they are more effective, and will provide analysis with specific context of a domain in cases where conventional methods are not effective, due to their lack of scalability, extensibility and rigor of analysis.

Chapter 7

Future Directions

Work that has been presented in this thesis evolved as a consequence of our experiences with conventional parallel program performance visualization tools. It contributes the following to the area of performance analysis:

- A model for representation and processing of the performance trace data, which allows the performance analyst to experiment with the program behavior.
- Modeling of the performance data with a matrix makes the application of appropriate statistical analysis, image and signal processing, and various analytical modeling methods possible.
- Matrix model for performance data provides uniformity on the "functional level" to develop and present new methods for the purpose of program performance analysis. It allows the analyst to develop methods of any degree of complexity due to its rigorous mathematical structure. It benefits from the linear algebraic techniques.
- Transformations of the performance data to other representations enhances the efficacy of the analysis and visualization.
- Multiple-domain analysis approach allows the user to precisely analyze various aspects of program behavior in specific domains.
- The techniques presented here are scalable, extensible and generic enough to be effective for different types of analysis for a variety of applications.
- This work tries to make the methods used in analytical performance modeling accessible to programmers of parallel computer systems who typically rely on execution trace data and analysis based on visualization of this data.

There are several areas that were only touched upon in this thesis, and work remains to be done in those areas. One of the objectives was to identify these areas for the continuation of this work. Several future directions are discussed in the following sections.

7.1 Implementation as a Toolkit

Analysis methods presented and the results shown in Chapter 6 were obtained by a prototype implementation of an analysis tool. One of the immediate future goals of this work is to complete this prototype implementation in the form of a generic performance

analysis tool with an appropriate user interface. However, the idea is not to add another tool to a large number of existing performance analysis tools that rarely enjoy any success in terms of their acceptability by the users. Therefore, we have adopted a "toolkit approach" towards the development of an analysis tool based on the ideas presented in this thesis [120]. The different needs and preferences of users make it difficult to develop a tool that is all-encompassing. As tool users, we want ease of use, flexibility, extensibility and advanced capabilities of data analysis. As tool developers, we know that it is difficult to accomplish all these objectives within a single tool. Thus, we advocate a toolkit approach that combines conventional performance visualization tools and off-the-shelf commercial data analysis tools. The latter tools are typically more mature and are often more readily accepted by a user of a parallel computer system (perhaps a scientist or an engineer who might already have used such tools for data visualization).

Presently, we have selected Pablo [86] as the basis of the toolkit, due to its advanced features of data transformation and module development. The toolkit approach is built on Pablo's approach of configuring modules for a specific performance analysis task. Pablo's approach is shown in Figure 7-1. The toolkit approach is shown in Figure 7-2. The toolkit

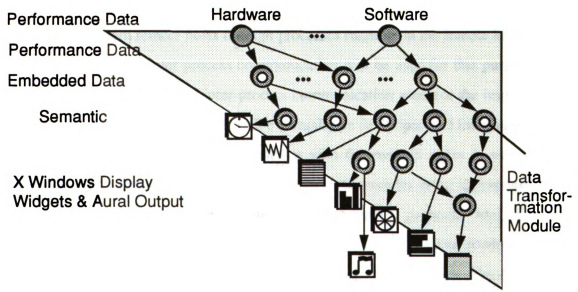


Figure 7-1. Performance Analysis Environment (based on Pablo).

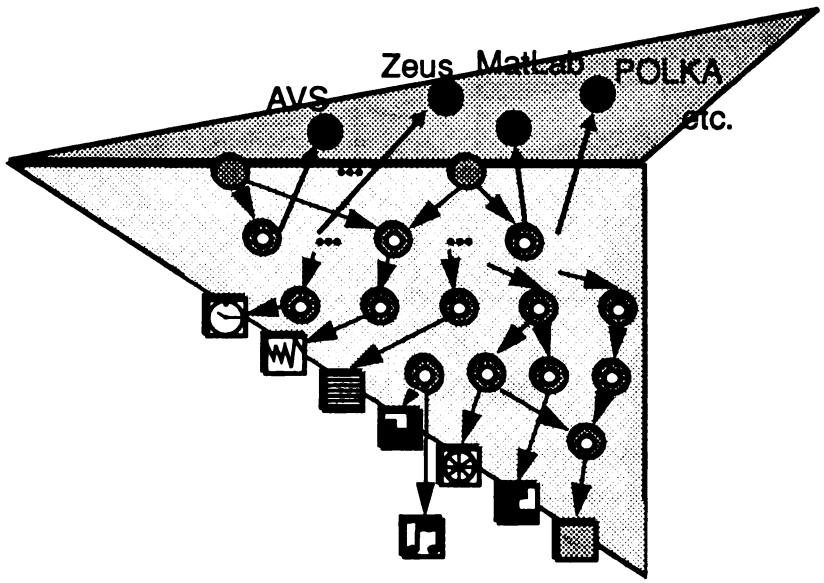


Figure 7-2. The toolkit approach.

builds on this modular approach by providing modules that interface Pablo to other tools. The user invokes predefined modules in Pablo to directly access a tool, or alternatively call specific functions of a tool.

We have used UNIX inter-process communication abstraction of *sockets* [110] for accessing tools from remote hosts through processes running on the remote hosts. A usual client-server model of inter-process communication will be used for this purpose. Figure 7-3 represents the approach of inter-process communication used for the implementation of the toolkit. The front-end module will eventually be developed and integrated in Pablo. It will consist of a user interface to interact with the user, in order to select specific performance trace data and to access specified analysis modules using appropriate tools. It will initiate the server processes on the hosts that have a particular tool. The server processes will wait for the client processes running from the front-end module to request connections (in response to user's requirement of using a specific tool). The driver of that particular tool (a server process) will exchange protocol information with the client process to signal the establishment of a connection. Then the client process can send data and control information to the server process which can pass that information on to the

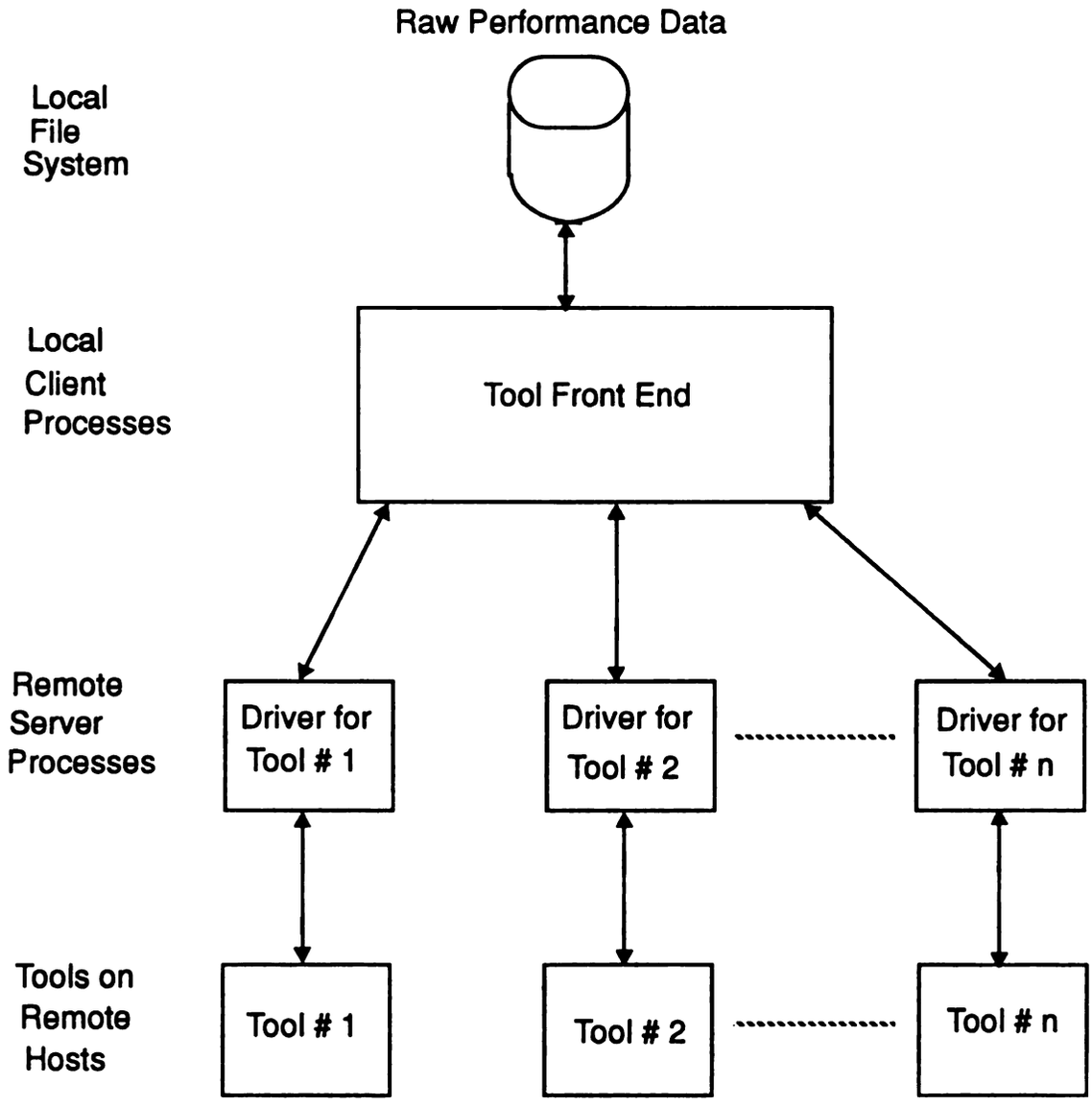


Figure 7-3. Practical implementation of the toolkit approach.

appropriate module of the tool (depending on the control information sent by the client). The tool will provide the appropriate results that will be displayed by the tool for the purpose of visualization. The server process will send the information about the successful (or otherwise) completion of the service that was requested from its tool. It will then start waiting for the new request from the client. The operation of all the tools will be synchronized by the common time-base at the client process, so that the displays of all the tools are updated synchronously.

7.2 Markov Modeling

We have already discussed in Chapter 5 that the Markov modeling approach can be used to model the behavior of the program. The program can be characterized by considering that the system can have a finite number of states. These states can define the state-space of the Markov chain that can model the behavior of the state transitions of the system either in the spatial domain or the event domain. Such a model can answer various questions about the behavior of the program. For instance, what are the *equivalence classes* of the state-space of the Markov chain? What are the closed equivalence classes that can determine *recurrent* or *transient* states? What are the probabilities of transition to a particular state that might be undesirable? An undesirable state might be a *dead-lock* state where system utilization will become zero and will remain there. All these questions are the basic questions that a Markov chain model can answer.

7.3 Modeling of Message-Passing

Message-passing among various processors of a multicomputer system during program execution is perhaps the most important factor that affects program performance. We have used metrics and patterns that resulted due to various message-passing events. However, we did not investigate message-passing characteristics such as volume of messages on the network at a given time during the system, the number of messages sent or received by each processor, average waiting time for a message to be received by the destination node after it was sent by the source node, etc. Queuing models are widely used for modeling the message-passing and network traffic characteristics. Execution trace data can be used with the multiple-domain analysis approach to test the validity of a particular model by comparing the model against the actual information. Such analysis can help the user evaluate the performance of a program from the perspective of inter-processor communication.

7.4 Performance Prediction

We are trying to develop an appropriate model for numeric-valued performance metrics in order to use the model to predict their future values. One such metric is the speed-up of a program with increasing numbers of processors. An appropriate model that could predict the future value of this metric would be very useful in practice. Many practical applications generate an enormous amount of event-trace data when instrumented due to a large number of events occurring during the execution of a program. We might be able to use information about appropriate parameters from executions on relatively smaller numbers of processors to determine the parameters needed for prediction of the metric on a larger number of processors. Such performance prediction can save the overheads of tracing programs on large numbers of processors and can still provide the required information about the characteristics of the program.

7.5 Dynamic Load Balancing

Work on system and program behavior modeling can be used to predict various system parameters dynamically. We had emphasized that the performance data models and transformations methods presented in this thesis can be used dynamically in real-time. This feature can be used to enable the operating system (or an application program) to predict the load-balance among the processors and/or the traffic on a particular segment of the network. This information can be used to dynamically re-schedule the jobs and to update the routing information to optimally manage the jobs running on the system. [53] use a Markov model for queuing jobs to determine an optimal job scheduling strategy. [102] describes a resource reclaiming algorithm for real-time multiprocessor systems. [11] formulates a network flow model for mesh and hypercube topologies with circuit-switched routing to dynamically transfer load from over-utilized nodes (sources) to under-utilized nodes (sinks). The performance data modeling and processing approach presented in this thesis can be extended to dynamically and optimally predict the loads on various nodes to assist the load sharing/balancing algorithm being used on a particular system.

7.6 Database Management Systems

There are usually two aspects of every system that depends heavily on data: data management and data analysis. In this thesis, data models have been concerned entirely with the performance analysis aspects and data management aspects were not considered. Nevertheless, data management techniques used in the realm of database management systems (DBMS) can only enhance the capabilities of data modeling approach and methods discussed here. Observed (performance) data can be considered as a hypothetical description of real-world (physical or logical) entities and objects. Data models are needed to represent, manage, and analyze the appropriate information regarding the real-world entities. Database management systems consider the "raw" data as input data or initial description of the states of the system modeled and to be analyzed by the data [112]. A database management system can be thought of as a program that helps formulate the appropriate transformations on the data to convert it to a suitable, information-bearing final form. This concept is described by Figure 7-4. There are various data models used by

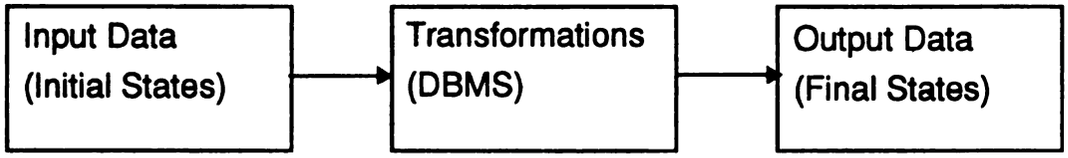


Figure 7-4. Operation of a database management system.

DBMSs, such as relational, functional, entity relationship models, etc. The purpose of these models is to provide a structure for representing data that could ensure a uniform interpretation of the same data by different users [112]. Functional data model particularly relies on functions that transform the data from one domain to another. The transformations presented in Chapter 4 were mostly expressed as a rule, as simple closed form transformations were not trivial to be defined. In such practical cases, use of functional data models to practically implement these transformations can greatly enhance the efficiency of the multiple-domain analysis approach discussed in this thesis.

Relational databases have been used for the purpose of program performance analysis [105]. However, if such techniques are used without an appropriate model for the analysis, "what if" type of questions can be answered but a detailed macroscopic and microscopic analysis of the program behavior might become difficult to construct by the user. However, we believe that the combination of the data modeling and analysis approaches used by DBMSs and analysis presented by this thesis can solve most of the performance data analysis problems. Work needs to be done to implement the transformations and analysis methods presented here, by using DBMSs.

These are only a few possible future directions of this work. The extensibility and rigor of the approach developed in this thesis enable it to potentially address performance analysis needs that might arise in the future but are not yet apparent.

List of References

List of References

[1] Abrams, M., N. Doraswamy, and A. Mathur, "Chitra: Visual Analysis of Parallel and Distributed Programs in the Time, Event and Frequency Domains," *IEEE Transactions on Parallel and Distributed Systems*, 3(6), November 1992.

[2] Ahluwalia, A. K. and M. Singhal, "Performance Analysis of the Communication Architecture of the Connection Machine," *IEEE Transactions on Parallel and Distributed Systems*, 3(6), November 1992.

[3] Almasi, George S. and Allan Gottlieb, *Highly Parallel Computing*, The Benjamin/ Cummings Publishing Company, Inc., 1989.

[4] Anton, Howard and Chris Rorres, *Elementary Linear Algebra — Applications Version*, John Wiley & Sons Inc., 1991. *IEEE Transactions on Parallel and Distributed Systems*, 3(6), November 1992.

[5] Aoki, Masanao, *State Space Modeling of Time Series*, Springer-Verlag, 1990.

[6] Appelbe, W. and C. McDowell, "Integrating Tools for Debugging and Developing Multitasking Programs," *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Madison*, Wisconsin, May 5-6, 1988.

[7] Assefi, Touraj, *Stochastic Processes and Estimation Theory with Applications*, John Wiley & Sons, Inc., 1979.

[8] AVS Module Reference Release 3.0, Stardent Computer Inc. April 1991.

[9] AVS User's Guide, Stardent Computer Inc. 1991.

[10] Beadle, P., C. Pommerell, and M. Annaratone, "K9: A Simulator of Distributed-Memory Parallel Processors," *Proc. of Supercomputing '89*, ACM Press, 1989.

[11] Bokhari, Shahid H., "A Network Flow Model for Load Balancing in Circuit-Switched Multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, 4(6), June 1993.

[12] Box, George P. and Gwilym M. Jenkins, *Time Series Analysis — Forecasting and Control*, Holden-Day Inc., 1976.

[13] Brown, Marc and John Hershberger, "Color and Sound in Algorithm Animation," *IEEE Computer*, December 1992.

[14] Buja, Andreas et al., "Interactive Data Visualization using Focusing and Linking," Proceedings of Visualization '91, 1991.

[15] Burger, Peter, Duncan Gillies, *Interactive Computer Graphics*, Addison-Wesley Publishing Company, Inc., 1989.

[16] Callahan, D. and J. Subhlok, "Static Analysis of Low-level Synchronization," *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Madison*, Wisconsin, May 5-6, 1988.

[17] Carlson, B. M, T. D. Wagner, L. W. Dowdy, and P. H. Worley, "Speedup Properties of Phases in the Execution Profile of Distributed Parallel Programs," Technical Report ORNL/TM-11900, Oak Ridge National Laboratory, August 1992.

[18] Carlson, Gordon E., *Signal and Linear System Analysis*, Houghton Mifflin Company, 1992.

[19] Casavant, Thomas L., "Tutorial: Software Tools for Visualization of Parallel and Distributed Programs and Systems," Department of Electrical and Computer Engineering, University of Iowa, September 1991.

[20] Casavant, Thomas L., "Tools and Methods for Visualization of Parallel Systems and Computations," *Journal of Parallel and Distributed Computing*, 18(2), June 1993.

[21] Chandy, K. M. and Jayadev Misra, *Parallel Program Design—A Foundation*, Addison-Wesley Publishing Company, Inc., 1988.

[22] Chang, Carl K., Young-Fu Chang, Lin Yang, Ching-Roung Chou, and Jong-Jeng Chen, "Modeling a Real-Time Multitasking System in Timed PQ Net," *IEEE Software*, March 1989.

[23] Cheng, Doreen Y., "A Survey of Parallel Programming Languages and Tools," Report RND-93-005, NASA Ames Research Center, March 1993.

[24] Clymer, John R., *Systems Analysis Using Simulation and Markov Models*, Prentice-Hall, Inc., 1990.

[25] Comerford, Richard, "Software on the Brink," *IEEE Spectrum*, 29(9), September 1992.

[26] Cooper, R. A., and A. J. Weekes, *Data, Models and Statistical Analysis*, Barens and Noble Books, 1983.

[27] Couch, Alva, "Graphical Representation of Program Performance on Hypercube Message-Passing Multiprocessors." *PhD. dissertation*, Department of Computer Science, Tufts University, April 1988.

[28] Couch, Alva, and Krumme, David W., "Projection, Pursuit, and the Triplex Tool Set for the NCUBE Multiprocessor," Department of Computer Science, Tufts University, November 1989.

[29] Couch, Alva, "Categories and Context in Scalable Execution Visualization," *Journal of Parallel and Distributed Computing*, 18(2), June 1993.

[30] DeGroot, Morris H., *Probability and Statistics*, Addison-Wesley Publishing Company, 1987.

[31] Dougharty, Edward E. and Charles R. Giardina, *Image Processing—Continuous to Discrete*, Volume 1, Prentice-Hall, Inc., 1987.

[32] Emrath, P. and D. Padua, "Automatic Detection of Nondeterminancy in Parallel Programs," *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Madison*, Wisconsin, May 5-6, 1988.

[33] Ferrari, Domenico, *Computer Systems Performance Evaluation*, Prentice-Hall, Inc., 1978.

[34] Ferrari, Domenico, "Considerations on the Insularity of Performance Evaluation," *IEEE Transactions on Software Engineering*, June 1986.

[35] Fineman, C. E. and P. J. Hontalas, "Selective Monitoring Using Performance Metric Predicates," *Proc. Scalable High-Perf. Comp. Conf.*, IEEE Comp. Soc., 1992.

[36] Fox, Geoffrey et. al., "Fortran D Language Specification," Technical Report, Syracuse University, January 1992.

[37] Fox, G., M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors: Volume 1—General Techniques and Regular Problems*, Prentice-Hall, Inc., 1988.

[38] Francioni, J., et al., "Working Group 3 Summary: Application Issues," in *Summary of Workshop on Par. Comp. Systems: Software Performance Tools*, ed. Hayes, Simmons, and Reed, Santa Fe, October 2–4, 1991.

[39] Francioni, J. and J. Jackson, "Breaking the Silence: Auralization of Parallel Program Behavior," *Journal of Parallel and Distributed Comp*uting, 18(2), June 1993.

[40] Funka-Lea, Cynthia A., Tasos D. Kontogioros, Robert J. T. Morris, and Larry D. Rubin, "Interactive Visual Modeling for Performance," *IEEE Software*, 8(5), September 1991.

[41] Gannon, D., "Predicting Performance: Spreadsheets and What-If Questions," presentation at Workshop on Parallel Computer Systems: Software Performance Tools, Santa Fe, October 2–4, 1991.

[42] Gardner, William A., I*ntroduction to Random Processes—With Applications to Signals and Systems*, Macmillan Publishing Company, 1986.

[43] Gehani, Narian, William D. Roome, *The Concurrent C Programming Language*, AT&T Bell Telephone Laboratories, Inc., 1989.

[44] Geist, G., M. Heath, B. Peyton, and P. Worley, "A Machine-Independent Communication Library," *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, Los Altos: Golden Gate Enterprises, 1990.

[45] Geist, G., M. Heath, B. Peyton, and P. Worley, "A User's Guide to PICL", Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, March 1991.

[46] Glenn, R. and D. Pryor, "Instrumentation for a Massively Parallel MIMD Application," *Journal of Parallel and Distributed Computing*, 12(3), July 1991.

[47] Graupe, D. "Tutorial: Applications of Signal and Image Processing to Medicine," *Proceedings of 1988 IEEE International Symposium on Circuits and Systems*, Espoo, Finland, June 7-9, 1988.

[48] Gustafson, John, Diane Rover, Stephen Elbert, and Michael Carter. "The Design of a Scalable, Fixed-time Computer Benchmark," *Journal of Parallel and Distributed Computing*, 12(4), August 1991.

[49] Hasegawa, T., H. Takagi, and Y. Takahashi, editors, *Performance of Distributed and Parallel Systems*, Elsevier Science Publishers B.V., 1989.

[50] Heath, Michael T. and Jennifer A. Etheridge, "Visualizing the Performance of Parallel Programs," *IEEE Software*, 8(5), September 1991, pp. 29–39.

[51] Hough, Alfred A. and Janice E. Cunny, "Belvedere: Prototype of a Pattern-Oriented Debugger for Highly Parallel Computation," *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 735–738, 1987.

[52] Hough, Alfred A. and Janice E. Cuny, "Perspective Views: A Technique for Enhancing Parallel Program Visualization," *Proc. 1990 Int. Conf. on Par. Proc.*, IEEE Comp. Soc., 1990.

[53] Huang, Jau-Hsiung and Leonard Kleinrock, "Performance Evaluation of Dynamic Sharing of Processors in Two-Stage Parallel Processing Systems," *IEEE Transactions on Parallel and Distributed Systems*, 4(3), March 1993.

[54] Humphrey, Watts S. and Nozer D. Singpurwalla, "Predicting (Individual) Software Productivity," *IEEE Transactions on Software Engineering*, 17(2), February 1991.

[55] Iazeolla, Giuseppe and Francesco Marinuzzi, "LISPACK—A Methodology and Tool for the Performance Analysis of Parallel Systems and Algorithms," *IEEE Transactions on Software Engineering*, 19(5), May 1993.

[56] *IEEE Transactions on Parallel and Distributed Systems*, special issue on measurement and evaluation, 3(2), November 1992.

[57] Jain, Anil K., *Fundamentals of Digital Image Processing*, Prentice-Hall, Inc., 1989.

[58] Jain, Raj, *The Art of Computer Systems Performance Analysis—Techniques for Experimental Design, Measurement, Simulation, and Modeling*, John Wiley & Sons, Inc., 1991.

[59] Kendall, Maurice and Keith Ord, *Time Series*, Edward Arnold, 1990.

[60] Knuth, D., *The Art of Computer Programming*, Addison-Wesley, 1981.

[61] Kraemer, Eileen and John T. Stasko, "The Visualization of Parallel Systems: An Overview," *Journal of Parallel and Distributed Computing*, 18(2), June 1993.

[62] Lange, F. H., *Correlation Techniques*, London Iliffe Books Ltd., 1967.

[63] Lavenberg, Stephen S., editor, *Computer Performance Modeling Handbook*, Academic Press, 1983.

[64] Law, Averill M. and W. D. Kelton, *Simulation Modeling and Analysis*, McGraw-Hill, Inc., 1991.

[65] Leblanc, Thomas J., John M. Mellor-Crummey, and Robert J. Fowler, "Analyzing Parallel Program Executions Using Multiple Views," *Journal of Parallel and Distributed Computing*, 9(2), June 1990.

[66] Li, Wei and Keshav Pingali, "Loop Transformations for NUMA Machines," *to appear in ACM SIGPLAN Notices*, 1993.

[67] MacDougall, M. H., *Simulating Computer Systems—Techniques and Tools*, The MIT Press, 1987.

[68] Malony, A. D., D. A. Reed, and H. A. G. Wijshoff, "Performance Measurement Intrusion and Perturbation Analysis," *IEEE Transactions on Parallel and Distributed Systems*, 3(4), July 1992.

[69] Marsan, M. A., G. Balbo, and G. Conte, *Performance Models of Multiprocessor Systems*, The MIT Press, 1986.

[70] MasPar MP-1 Hardware Manual, MasPar Computer Corporation, September 1990.

[71] *Matlab User's Guide*, The Math Works Inc. August 1992.

[72] Mattuck, Richard D., *A Guide to Feynman Diagrams in the Many-Body Problem*, 1965.

[73] Miller, Barton P., Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System," *IEEE Transactions on Parallel and Distributed Systems*, 1(2), April 1990, pp. 206–217.

[74] Miller, Barton P., "What to Draw? When to Draw? An Essay on Parallel Program Visualization," *Journal of Parallel and Distributed Computing*, 18(2), June 1993.

[75] Mullin, Lenore M. R., "A Mathematics of Arrays," *Ph.D. Dissertation*, Department of Computer and Information Science, Syracuse University, December 1988.

[76] nCUBE 2 Processor Manual, Rev. 2.0, nCUBE Corporation, December 1990.

[77] Nichols, Kathleen and Paul W. Oman, "Navigating Complexity to Achieve High Performance," *IEEE Software*, 8(5), September 1991.

[78] Noor, Ahmed K. and Samuel L. Venneri, "A Perspective on Computational Structures Technology," *IEEE Computer*, 26(10), October 1993.

[79] Pancake, Cherri M., "Software Support for Parallel Computing: Where Are We Headed?", *Comm. ACM*, Nov. 1991.

[80] Pancake, Cherri M. and Sue Utter, "Models for Visualization in Parallel Debuggers," *Supercomputing '89*, November 1989.

[81] Papoulis, Athanasios, *Signal Analysis*, McGraw-Hill, Inc., 1977.

[82] Peebles, Peyton Z., *Probability, Random Variables, and Random Signal Principles*, McGraw-Hill, Inc., 1993.

[83] Pfeiffer, Paul E., *Concepts of Probability Theory*, Dover Publications, Inc., 1978.

[84] Proakis, John G. and Dimitris G. Manolakis, *Digital Signal Processing--Principles, Algorithms, and Applications*, Macmillan Publishing Company, 1992.

[85] Rabiner, L. R. and R. W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, 1978.

[86] Reed, Daniel A., Ruth A. Aydt, Tara M. Madhyastha, Roger J. Noe, Keith A. Shields, Bradley W. Schwartz, "The Pablo Performance Analysis Environment," Department of Computer Science, University of Illinois, 1992.

[87] Reed, D., "Scalable Performance Analysis and Language Issues", presentation at Workshop on Parallel Computer Systems, Keystone, April 3–5, 1993.

[88] Resnick, Sidney I., *Adventures in Stochastic Processes*, Birkhauser, 1992.

[89] Rover, Diane T. "Visualization of Program Performance on Concurrent Computers," *Ph.D. Dissertation*, Dept. of Electrical Engineering and Computer Engineering, Iowa State University, December 1989.

[90] Rover, Diane, Michael Carter, and John Gustafson. "Performance Visualization of SLALOM," *Proceedings of the Sixth Distributed Memory Computing Conference*, IEEE Computer Society, New York, 1991.

[91] Rover, Diane T., "A Performance Visualization Paradigm for Data-Parallel Computing," *Proceedings of the 25th Hawaii International Conference on System Sciences*, New York: IEEE Computer Society, 1992, pp. 146–160.

[92] Rover, Diane T., Joan F. Francioni, and Markus Doetsch, " A Survey of PICL and ParaGraph Users (1992)," Technical Report TR-MSU-EE-SCSL-01193, Department of Elec. Eng., Michigan State University, February, 1993.

[93] Rover, Diane T., A. Waheed, and M. Doetsch, "Advanced Methods of Performance Data Processing and Analysis," *Proceedings of the Seventh International Parallel Processing Symposium*, Newport Beach, April 13-16, 1993.

[94] Rover, Diane T. and A. Waheed, "Visualization, Auralization, and Analysis Methods," *Third ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, May 17-18, 1993.

[95] Rover, Diane T. and Charles T. Wright, "Visualizing the performance of SPMD and Data-Parallel Programs," *Journal of Parallel and Distributed Computing*, 18(2), June 1993.

[96] Rugh, Wilson J., *Linear System Theory*, Prentice-Hall, 1993.

[97] Ruschitzka, M., editor, *Computer Systems: Performance and Simulation*, Elsevier Science Publishers B.V., 1986.

[98] Russ, John C., *The Image Processing Handbook*, CRC Press, Inc., 1992.

[99] Sandquist, Gary M., *Introduction to system science*, Prentice-Hall, 1985.

[100] Sauer, Charles H. and K. M. Chandy, *Computer Systems Performance Modeling*, Prentice-Hall, Inc., 1981.

[101] Schilling, Robert J. and Hua Lee, *Engineering Analysis—A Vector Space Approach*, John Wiley & Sons, 1988.

[102] Shen, Chia, Krithi Ramamritham, and John A. Stankovic, "Resource Reclaiming in Multiprocessor Real-Time Systems," *IEEE Transactions on Parallel and Distributed Systems*, 4(4), April 1993.

[103] Simmons, M., and R. Koskela, editors, *Performance Instrumentation and Visualization*, ACM & Addison-Wesley, 1990.

[104] Simmons, M., R. Koskela, I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, ACM & Addison-Wesley, 1989.

[105] Snodgrass, R., "A Relational Approach to Monitoring Complex Systems," *ACM Transactions on Computer Systems*, 6, No. 2, May 1988.

[106] Sorensen, Erling V., Jens Nordahl, and Niels H. Hansen, "From CSP Models to Markov Models," *IEEE Transactions on Software Engineering*, 19(6), June 1993.

[107] Stallings, William, *Data and Computer Communications*, Macmillan Publishing Company, 1991.

[108] Stanat, Donald F. and David F. McAllister, *Discrete Mathematics in Computer Science*, Prentice-Hall, Inc., 1977.

[109] Stasko, John, and Eileen Kraemer, "A Methodology for Building Application-Specific Visualization of Parallel Programs," *Journal of Parallel and Distributed Computing*, 18(2), June 1993.

[110] Stevens, W. R., *UNIX Network Programming*, Prentice-Hall, Inc., 1990.

[111] Stone, H. *High-Performance Computer Architecture*, Addison-Wesley, 1987.

[112] Thompson, J. P., *Data with semantics: data models and data management*, Van Nostrand Reinhold, 1989.

[113] Timothy, LaMar K., *State space analysis: an introduction*, McGraw-Hill, 1968.

[114] Tsichritzis, Dionysios C., *Data models*, Prentice-Hall, 1982.

[115] Tsuei, T. F. and M. K. Vernon, "A Multiprocessor Bus Design Model Validated by System Measurement," *IEEE Transactions on Parallel and Distributed Systems*, 3(6), November 1992.

[116] Tufte, Edward R., *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, Connecticut, 1983.

[117] Tufte, Edward R., *Envisioning Information*, Graphics Press, Cheshire, Connecticut, 1990.

[118] Utter-Honig, Sue and Cherri M. Pancake, "Graphical Animation of Parallel Fortran Programs," *Supercomputing '91*, November 18 - 22, 1991.

[119] Waheed, A., B. Kronmuller, and D. T. Rover, "A Matrix Approach to Performance Data Modeling, Analysis and Visualization," accepted in *International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, Durham, North Carolina, Jan. 31-Feb. 2, 1994.

[120] Waheed, A., B. Kronmuller, and D. T. Rover, "A Toolkit for Advanced Performance Analysis," accepted in *International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94) Tools Fair*, Durham, North Carolina, Jan. 31– Feb. 2, 1994.

[121] Waheed, A. and D. T. Rover, "Performance Visualization of Parallel Programs," *Visualization '93*, San Jose, California, Oct. 25-29, 1993.

[122] Waheed, A. and D. T. Rover, "Performance Visualization and Analysis Using Multiple Domains," Technical Report TR-MSU-EE-SCSL-013-93, Department of Electrical Engineering, Michigan State University, March 1993.

[123] Waheed, A. and D. T. Rover, "Estimation of Repetitive Patterns and Loops in Parallel Programs," Technical Report TR-MSU-EE-SCSL-014-93, Department of Electrical Engineering, Michigan State University, March 1993.

[124] Waheed, A. and D. T. Rover, "Patterns of Parallel Programs," Technical Report TR-MSU-EE-SCSL-015-93, Department of Electrical Engineering, Michigan State University, March 1993.

[125] Ward, Matthew, James M. Coggins, Juan Herman, Theo Paulidis, Norman Wittels. "Graphics and Imaging: Trends Towards Unification?," *Proceedings of Visualization '90*, San Francisco, California, Oct. 23–26, 1990.

[126] Wegman, Edward J. and James G. Smith, editors, *Statistical Signal Processing*, Marcel Dekker, Inc., 1984.

[127] Widrow, Bernard and Samuel D. Stearns, *Adaptive Signal Processing*, Prentice-Hall, Inc., 1985.

[128] Wilcke, W. W., et al., "The IBM Victor Multiprocessor Project," *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, Los Altos: Golden Gate Enterprises, 1990.

[129] Yan, Jerry C., "Performance Instrumentation and Visualization Using AXE," in *Performance Instrumentation and Visualization*, Simmons and Koskela, ed., ACM & Addison-Wesley, 1990.

[130] Zadeh, Lotfi Asker, *Linear system theory; the state space approach*, McGraw-Hill, 1963.

Color Plates

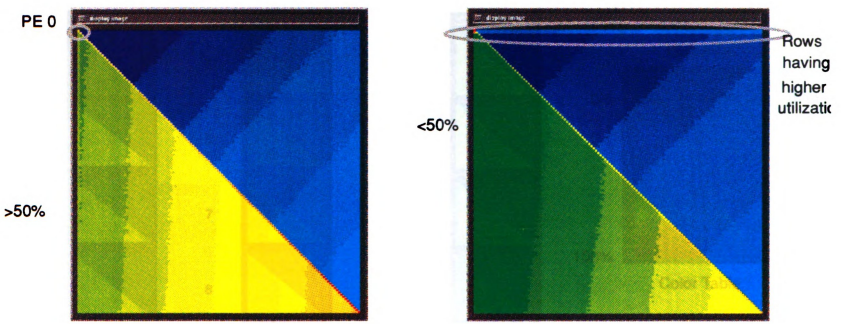


Figure 6-4. Performance images of PE utilization: even and uneven distributions.

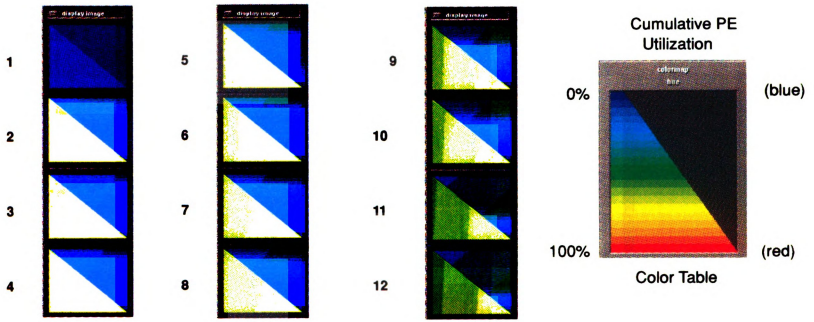


Figure 6-5. Performance images of GE in SLALOM (Series of 12 Snapshots Over Time).

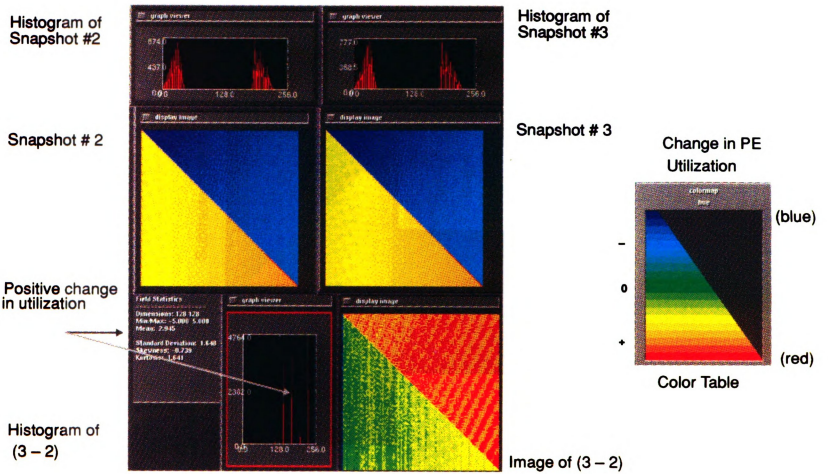


Figure 6-6. Change detection via image subtraction.

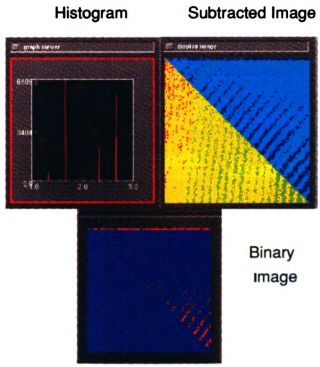


Figure 6-7. Analysis by small multiples using image subtraction and binary thresholding.

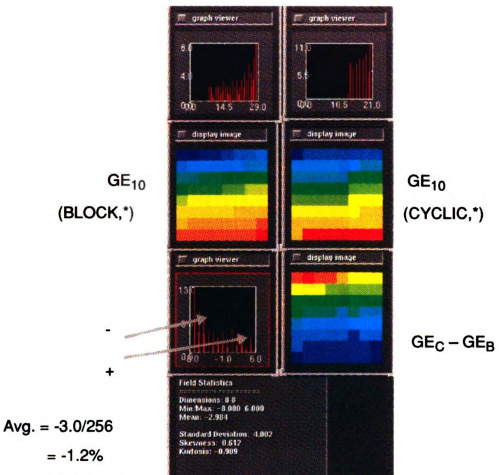


Figure 6-8. Comparison among GE phases of different data distributions via template matching.

BS_{10}

GE_{10}

$GE_{10} - BS_{10}$

Avg. = 13.7/256
= 5.4%
(increase)

Figure 6-9. Comparison among GE and BS phases of linear solver.

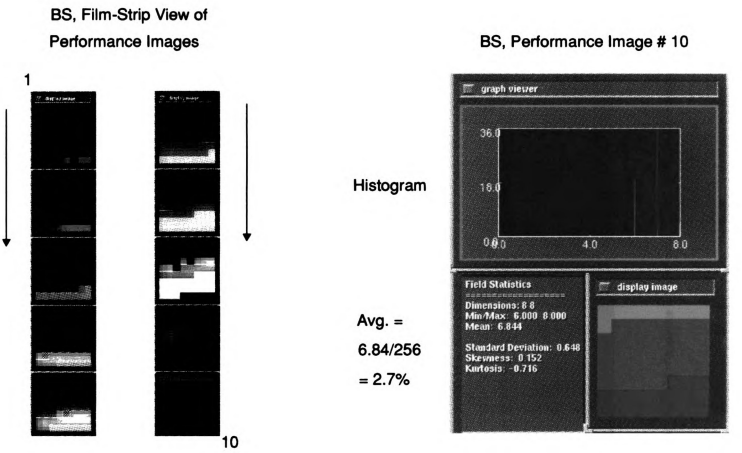
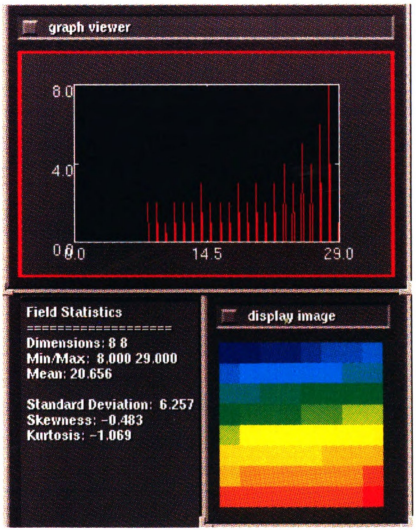


Figure 6-28. Performance images taken after equal intervals during BS phase and analysis of last image.

(BLOCK,*), GE

Histogram



Avg. = 20.65/256 = 7.8%

Figure 6-31. Performance image from GE phase with its histogram.